



SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Call Identifier:	FP7-ICT-2011-7
Project Number:	287305
Project Acronym:	OpenIoT
Project Title:	Open source blueprint for large scale self-organizing cloud environments for IoT applications

D5.2.2 Privacy and Security Framework b

Document Id:	OpenIoT-D522-140819-Draft
File Name:	OpenIoT-D522-140819-Draft.pdf
Document reference:	Deliverable 5.2.2
Version:	Draft
Editor:	Jean-Paul Calbimonte
Organisation:	EPFL
Date:	2014 / 08 / 19
Document type:	Deliverable (Prototype)
Dissemination level:	PU (Public)

Copyright © 2014 OpenIoT Consortium: NUIG-National University of Ireland Galway, Ireland; EPFL - Ecole Polytechnique Fédérale de Lausanne, Switzerland; Fraunhofer Institute IOSB, Germany; AIT - Athens Information Technology, Greece; CSIRO - Commonwealth Scientific and Industrial Research Organization, Australia; SENSAP Systems S.A., Greece; AcrossLimits, Malta; UniZ-FER University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia. Project co-funded by the European Commission within FP7 Program.

PROPRIETARY RIGHTS STATEMENT

This document contains information, which is proprietary to the OpenIoT Consortium.
Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the consortium.

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V201	Jean-Paul Calbimonte	EPFL	2014/08/19	ToC Revised based on D521
V202	Mehdi Riahi	EPFL	2014/08/23	Security framework implementation.
V203	Jean-Paul Calbimonte	EPFL	2014/08/25	Revision of contributions, adapting and harmonizing previous contents.
TR	Arkady Zaslavsky	CSIRO	2014/09/22	TR
V204	Jean-Paul Calbimonte	EPFL	2014/09/23	Comments after TR
TR	Arkady Zaslavsky	CSIRO	2014/09/24	TR & ToC adjustment

TABLE OF CONTENTS

1	INTRODUCTION	8
1.1	Scope	8
1.2	Audience	8
1.3	Methodology	8
1.4	Related Documents	9
1.5	Structure	9
2	SECURE MESSAGING.....	10
2.1	Message digest	10
2.2	Private key cryptography	10
2.3	Public key cryptography	12
2.4	Digital signatures	13
2.5	Digital certificates	14
2.5.1	Keytool and keystore	15
2.5.2	CertPath API	15
3	SECURITY PROTOCOLS.....	17
3.1	IEEE802.15.4.....	17
3.2	IPsec.....	17
3.3	TSL	18
3.4	HTTPS	20
4	ACCESS CONTROL.....	22
4.1	Fundamental concepts	22
4.2	Lattice-Based Access Control Models	24
4.2.1	Information flow policy	24
4.3	Access control models-based on information flow	25
4.3.1	Mandatory access policy	26
4.3.2	Confidentiality	26
4.3.3	Integrity	26
4.3.4	Combined	27
5	THE SECURITY ARCHITECTURE IN OPENIOT	28
5.1	Overview of the architecture	28
5.2	Access matrix	29
5.3	Mandatory access rules.....	30
5.4	Security protocol stack	30
6	AUTHORISATION FRAMEWORK.....	31
6.1	Introduction	31
6.2	Roles.....	32
6.3	OAuth Protocol Flow.....	33
6.3.1	Authorisation Grant.....	34
6.3.2	Authorisation Code	34
6.3.3	Implicit.....	35
6.3.4	Resource Owner Password Credentials	35
6.3.5	Client Credentials	36
6.4	Access Token.....	36
6.5	Refresh Token.....	36
6.6	TLS Version	38
6.7	HTTP Redirections	38
6.8	Interoperability	38
6.9	Security considerations	38

7	TRUSTWORTHINESS OF SENSOR READINGS	40
7.1	Notation	40
7.2	Problem definition	40
7.3	Overview of the method.....	41
7.4	Algorithm.....	42
7.5	Trust-Module architecture for IoT	43
8	IMPLEMENTATION OF THE SECURITY FRAMEWORK.....	45
8.1	General Authentication and Authorisation Approach	45
8.1.1	<i>Authentication</i>	46
8.1.2	<i>Authorisation</i>	47
8.2	Openlot CAS.....	49
8.2.1	<i>Configuration</i>	49
8.2.2	<i>JBoss Configuration</i>	50
8.2.2.1	OpenIoT CAS Configuration.....	50
8.2.3	<i>Deployment</i>	52
8.2.4	<i>Managing Services in OpenIoT CAS</i>	52
8.3	Security Management Console	52
8.3.1	<i>Configuration</i>	52
8.3.2	<i>Deployment</i>	53
8.3.3	<i>Use</i>	53
8.3.3.1	Managing Services	54
8.3.3.2	Managing Permissions.....	55
8.3.3.3	Managing Roles	57
8.3.3.4	User Management	57
8.3.3.5	User Registration	58
8.3.3.6	Demo Services.....	58
8.3.3.7	Management Console Permissions	59
8.4	Security Client.....	60
8.4.1	<i>Deployment</i>	60
8.4.2	<i>Maven Dependencies</i>	60
8.4.3	<i>Usage for Web Applications</i>	60
8.4.3.1	Shiro Webapp Configuration	60
8.4.3.2	Manual Redirection	63
8.4.3.3	Checking for Permissions and Roles	64
8.4.3.4	Tag libraries.....	64
8.4.4	<i>RESTful Usage</i>	65
8.4.4.1	Configuration.....	65
8.4.4.2	SSL Troubleshooting.....	67
8.4.4.3	Authentication and Authorisation.....	67
8.4.4.4	Sample Restful Usage	69
8.4.5	<i>Cache Configuration</i>	69
8.4.6	<i>Token Expiry</i>	69
9	INTEGRATION OF OPENIOT COMPONENTS WITH THE SECURITY FRAMEWORK.....	71
9.1	LSM Server	71
9.2	X-GSN	71
9.3	Scheduler.....	72
9.4	SDUM	73
9.5	Request Definition	73
9.6	Request Presentation	74
10	CONCLUSIONS	76
11	REFERENCES	77

LIST OF FIGURES

FIGURE 1. SECURITY ARCHITECTURE IN OPENIoT.....	28
FIGURE 2. ABSTRACT PROTOCOL FLOW.....	33
FIGURE 3. AUTHORISATION CODE FLOW.	34
FIGURE 4. REFRESHING AN EXPIRED ACCESS TOKEN.	37
FIGURE 5. TRUST OF SENSOR STREAM REPRESENTATIONS.	42
FIGURE 6. TRUST-MODULE IN OPENIoT.....	44
FIGURE 7. HOME PAGE OF MANAGEMENT CONSOLE APPLICATION WITH “LOG IN” LINK	46
FIGURE 8. OPENIoT CAS LOGIN PAGE.....	47
FIGURE 9. OPENIoT CAS AUTHORISATION PAGE.....	47
FIGURE 10. HOME PAGE OF MANAGEMENT CONSOLE APPLICATION AFTER USER IS AUTHENTICATED	47
FIGURE 11. SEQUENCE DIAGRAM OF A SAMPLE ACCESS CONTROL	48
FIGURE 12. MANAGING SERVICES IN MANAGEMENT CONSOLE APPLICATION.....	54
FIGURE 13. ADDING/EDITING WEB APPLICATION SERVICES	55
FIGURE 14. ADDING/EDITING REST SERVICES	55
FIGURE 15. PERMISSION MANAGEMENT SECTION OF MANAGEMENT CONSOLE	56
FIGURE 16. THE LIST OF PERMISSIONS DEFINED FOR THE "LSM-SERVER" SERVICE.....	56
FIGURE 17. ADDING NEW PERMISSION FOR THE SELECTED SERVICE	57
FIGURE 18. ROLE MANAGEMENT IN MANAGEMENT CONSOLE APPLICATION.....	57
FIGURE 19. USER MANAGEMENT IN MANAGEMENT CONSOLE APPLICATION	58
FIGURE 20. USER REGISTRATION PAGE.....	59

LIST OF TABLES

TABLE 1. AN EXAMPLE ACCESS MATRIX	23
TABLE 2. A FRAGMENT OF THE ACCESS MATRIX IN OPENIoT.	29
TABLE 3. SECURITY CLASSES AND ROLES IN OPENIoT.....	30
TABLE 4. THE SECURITY PROTOCOL STACK IN OPENIoT.....	30
TABLE 5. PERMISSIONS FOR WORKING WITH THE MANAGEMENT CONSOLE APPLICATION.....	59
TABLE 6. JSF TAGS PROVIDED BY THE THE SECURITY CLIENT MODULE	65

TERMS AND ACRONYMS

AIT	Athens Information Technology
CAS	Central Authorisation Server
CMC	Config/Monitor Console
DERI	Digital Enterprise Research Institute
EPFL	Ecole Polytechnique Fédérale de Lausanne
EU	European Union
FP7	7th Framework Programme for Research and Technological Development
ICO	Internet Connected Object
IoT	Internet of Things
JDK	Java Development Kit
LSM	Sensor Data Cloud Database
PS	Physical sensor
SCH	Scheduler
SDUM	Service Delivery and Utility Manager
TM	Trust Module
VS	Virtual sensor
X-GSN	Extended Global Sensors Network

1 INTRODUCTION

1.1 Scope

This deliverable describes the *Security and Privacy Framework* of the OpenIoT platform, including details about its design and implementation. The aim of this framework is to ensure that Internet-Connected Objects (ICO) contributing to the OpenIoT platform, its internal modules and external applications will communicate through secured IoT data interfaces (according to the target security/confidentiality level specified by the user).

Moreover we show the feasibility of this security module in the implemented prototype, which is an integral part of the OpenIoT platform. In particular we describe the implementation of the *Central Authorisation Server* (CAS), the *Security Management console*, the *Security Client*, and the integration of the security framework in the core modules of the platform.

This deliverable is the second and final of a series of two describing the overall privacy and security functional framework of OpenIoT.

1.2 Audience

This privacy and security framework report and prototype deliverable addresses the following audiences:

- **Technical Developers**, for sharpening the privacy and security framework planning and development; We provide technical details about the integration of the security framework in the OpenIoT reference implementation, as well as guidelines to integrate with new modules or applications.
- **Business Developers**, by taking into account the described privacy and security components in order to design, implement and fine-tune the framework among other components of the OpenIoT architecture, semantic infrastructure, management framework, middleware and proof-of-concept applications in OpenIoT;
- **The European Commission**, in order to assess the OpenIoT progress regarding privacy and security perspectives.

1.3 Methodology

Security, privacy and trust issues in Internet of Things (IoT) are of fundamental importance and guaranteeing the highest standards in those respects is necessary to advance the application of (IoT). First of all, any future large-scale deployment of OpenIoT will only be possible if it complies with corresponding legal security/privacy and reliability requirements (e.g., recent related EU regulations on privacy and security). Clearly, apart from the legal requirements, a lack of proper security/privacy and trust mechanisms in OpenIoT may lead to disastrous consequences as a result of hacking OpenIoT-based systems. As an example, consider pacemakers or insulin pumps that start functioning differently (perhaps, maliciously). As another example, consider smart meters, where as a result of hacking the bills start going up. One of the biggest challenges in designing a proper security/privacy and trust mechanism in OpenIoT is the modular and heterogeneous composition of system elements in most

existing IoT infrastructures, having very differing security/privacy and reliability standards (e.g., fixed versus mobile devices).

1.4 Related Documents

It is assumed that the reader is familiar with the OpenIoT technology as described in the following deliverables:

- D2.2 OpenIoT Platform Requirements and Technical Specifications.
- D2.3 OpenIoT Detailed Architecture and Proof-of-Concept Specifications.
- D3.1.1 Semantic Representations of Internet-Connected Objects a.
- D3.1.2 Semantic Representations of Internet-Connected Objects b.
- D3.2 Semantic Communication Protocols (Object/Object and People/Object).
- D4.1 Service Delivery Environment Formulation Strategies.
- D4.3.2 Core OpenIoT Middleware Platform b
- D5.2.1 OpenIoT Privacy and Security Framework a

1.5 Structure

This document describes first the foundations of the OpenIoT security framework, including secure protocols for information exchange (secure messaging), access control, authentication and authorisation. The challenges in designing the authentication and authorisation model for OpenIoT stem from the need to accommodate differing requirements from the distributed components of the OpenIoT platform. Then we describe in detail the implementation of the Security Framework, including the OpenIoT CAS server, the Security Client, and the Management console. This document presents the final version of the security/privacy and trust specification (deliverable), including the description of the final implementation.

The content of this document is as follows: Section 2 reviews the theory of secure messaging, Section 3 reviews applicable security protocols, Section 4 reviews the theory of access control (authentication and authorisation), Section 5 presents the proposed security module in OpenIoT, Section 6 presents details of the proposed authorisation framework, Section 7 presents trust mechanisms for IoT, Section 8 discusses the full implementation of the security framework and Section 9 presents the details of the integration with all the core modules of the OpenIoT platform.

2 SECURE MESSAGING

In this section we review fundamental security concepts that are designed to provide secure messaging. The challenges in this area include the necessity to send messages preventing that an eavesdropper can intercept and read the information embedded in the messages. They are also concerned with mechanisms to obfuscate or encrypt the message content in such a way that the receiver can guarantee that the message has not been modified by a third party and that it originates from the sender it claims to be.

These concepts and frameworks have been widely implemented in different types of systems and platforms, and they are supported in the Java platform [13], in which the OpenIoT platform has been developed. The presented concepts are crucial in understanding secure protocols discussed in the following sections. In particular we review the following ones:

- ✧ **Message digests.** Technology that ensures the integrity of a transmitted message, along with message authentication codes.
- ✧ **Private key encryption.** A mechanism designed to ensure the confidentiality of a message, based on a private key shared by communicating parties.
- ✧ **Public key encryption.** A mechanism that allows two parties to share secret messages without prior agreement on secret keys.
- ✧ **Digital signatures.** A pattern that identifies the other party's message as coming from the appropriate person.
- ✧ **Digital certificates.** A technology that adds another level of security to digital signatures by having the message certified by a third-party authority.

2.1 Message digest

A message digest is a function that generates a unique number or code that represents the original message, ensuring the integrity of that message. More specifically, message digests take a message as input and generate a block of bits, usually several hundred bits long that represents the *fingerprint* of the message. A small change in the message (say, by an interloper or eavesdropper) creates a noticeable change in the fingerprint.

Message-digest functions are one-way functions, so it is generally simple to generate the fingerprint from the message, but quite difficult to generate a message for a given fingerprint.

Message digests can be weak or strong. A checksum, which is the XOR of all the bytes of a message, is an example of a weak message-digest function. It is easy to modify one byte to generate any desired checksum fingerprint. Most strong functions use hashing. A 1-bit change in the message leads to a massive change in the fingerprint (ideally, 50 percent of the fingerprint bits change).

If a key is used as part of the message-digest generation, the algorithm is known as a message-authentication code.

2.2 Private key cryptography

Message digests may ensure integrity of a message, but they can't be used to

ensure the confidentiality of a message. For that, private key cryptography has been designed to exchange private messages [15,18].

Consider this scenario: Alice and Bob each have a shared key that only they know and they agree to use a common cryptographic algorithm, or cipher. When Alice wants to send a message to Bob, she encrypts the original message, known as *plaintext*, to create *ciphertext* and then sends the ciphertext to Bob. Bob receives the ciphertext from Alice and decrypts the ciphertext with his private key to re-create the original plaintext message. If Eve the eavesdropper is listening in on the communication, she hears only the ciphertext, so the confidentiality of the message is preserved.

The JDK (Java Development Kit) supports the following private key algorithms [14]:

- ⤴ **DES.** DES (Data Encryption Standard) was invented by IBM in the 1970s and adopted by the U.S. government as a standard. It is a 56-bit block cipher [19].
- ⤴ **TripleDES.** This algorithm is used to deal with the growing weakness of a 56-bit key while leveraging DES technology by running plaintext through the DES algorithm three times, with two keys, giving an effective key strength of 112 bits. TripleDES is sometimes known as DESede (for encrypt, decrypt, and encrypt, which are the three phases).
- ⤴ **AES.** AES (Advanced Encryption Standard) replaces DES as the U.S. Standard. It is a 128-bit block cipher with key lengths of 128, 192, or 256 bits [20].
- ⤴ **RC2, RC4, and RC5.** These are algorithms from a leading encryption security company, RSA Security.
- ⤴ **Blowfish.** This algorithm is a block cipher with variable key lengths from 32 to 448 bits (in multiples of 8), and was designed for efficient implementation in software for microprocessors.
- ⤴ **PBE.** PBE (Password Based Encryption) can be used in combination with a variety of message digest and private key algorithms. The Cipher class manipulates private key algorithms using a key produced by the KeyGenerator class.

For example, the following code allows ciphering a message using DES:

```
Cipher c = Cipher.getInstance("DES");
c.init(Cipher.ENCRYPT_MODE, sKey);
```

Similarly to decode the message:

```
c.init(Cipher.DECRYPT_MODE, sKey);
try {
    String s = (String)so.getObject(c);
} catch (Exception e) {
    // do something
}
```

The Java security package also includes private key and keystore API methods. For example to create a private key:

```
byte[] desKeyData = { (byte)0x01, (byte)0x02, (byte)0x03,
    (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, (byte)0x08 };
```

```
DESKeySpec desKeySpec = new DESKeySpec(desKeyData);
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
```

2.3 Public key cryptography

Private key cryptography suffers from a major drawback, which is the mechanism by which the related parties exchange keys. For instance if Alice generates a private key, she has to send it to Bob, but it is sensitive information so the key itself should be encrypted. However, keys have not been exchanged to perform the encryption and we end up in a chicken-egg situation.

Public key cryptography [15,16] solves the problem of encrypting messages between two parties without prior agreement on the key. In fact, in public key cryptography, Alice and Bob have different keys (key pairs) which are related. A message encrypted with one key can only be decrypted with the other and vice-versa. One key is *private* and must not be shared with anyone. The other key is *public* and can be shared with anyone. When Alice wants to send a secure message to Bob, she encrypts the message using Bob's public key and sends the result to Bob. Bob uses his private key to decrypt the message. When Bob wants to send a secure message to Alice, he encrypts the message using Alice's public key and sends the result to Alice. Alice uses her private key to decrypt the message. Eve can eavesdrop on both public keys and the encrypted messages, but she cannot decrypt the messages because she does not have either of the private keys.

The public and private keys are generated as a pair and need longer lengths than the equivalent-strength private key encryption keys. Typical key lengths for the RSA algorithm are 1,024 bits. It is not feasible to derive one member of the key pair from the other.

Public key encryption is slow compared to the private key approaches (100 to 1,000 times slower than private key encryption), so a hybrid technique is usually used in practice. Public key encryption is used to distribute a private key, known as a session key, to another party, and then private key encryption using that private session key is used for the bulk of the message encryption.

The following two algorithms are used in public key encryption:

- ✧ **RSA.** This algorithm is the most popular public key cipher, but it's not supported in JDK and one needs to use a third-party library like BouncyCastle to get this support.
- ✧ **Diffie-Hellman.** This algorithm is technically known as a key-agreement algorithm. It cannot be used for encryption, but can be used to allow two parties to derive a secret key by sharing information over a public channel. This key can then be used for private key encryption [21].

As an example, the following code shows how to generate a key-pair in Java of size 1024 [14]:

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
random.setSeed(userSeed);
keyGen.initialize(1024, random);
```

The SecureRandom object provides the randomness for the generation of the keys,

and can be customized. Then the keyGen object is ready to generate new key-pairs:

```
KeyPair pair = keyGen.generateKeyPair();
```

2.4 Digital signatures

Digital signatures are typically used for the task of determining the identification of parties that exchange messages. The problem arises from possible impersonation attacks, where the receiver wants to know if the sender of an encrypted message really is the one who sent the message [15]. The public key message exchange described before does not describe or provide a mechanism to perform this type of check: e.g. how can Bob prove that the message really came from Alice? Eve (an eavesdropper) could have substituted her public key for Alice's, and then Bob would be exchanging messages with Eve thinking she was Alice. This is known as a *Man-in-the-Middle attack*. This problem can be solved by using a digital signature, i.e., a bit pattern that proves that a message came from a given party.

One way of implementing a digital signature is using the reverse of the public key process. Instead of encrypting with a public key and decrypting with a private key, the private key is used by a sender to sign a message and the recipient uses the sender's public key to decrypt the message. Because only the sender knows the private key, the recipient can be sure that the message really came from the sender.

Therefore, the message digest, not the entire message, is the bit stream that is signed by the private key. So, if Alice wants to send Bob a signed message, she generates the message digest of the message and signs it with her private key. She sends the message (in the clear) and the signed message digest to Bob. Bob decrypts the signed message digest with Alice's public key and computes the message digest from the cleartext message and checks that the two digests match. If they do, Bob can be sure the message came from Alice.

Note that digital signatures do not provide encryption of the message, so encryption techniques must be used in conjunction with signatures if confidentiality is also needed. We can use the RSA algorithm for both digital signatures and encryption. A U.S. standard called DSA (Digital Signature Algorithm) can be used for digital signatures, but not for encryption.

The JDK supports the following digital signature algorithms:

- ⤴ MD2/RSA.
- ⤴ MD5/RSA.
- ⤴ SHA1/DSA.
- ⤴ SHA1/RSA.

We show an example of how to create a signature using the Java cryptography API [14]. We first create a signature object with DSA and SHA:

```
Signature dsa = Signature.getInstance("SHA1withDSA");
```

Next, using the key pair generated beforehand (e.g. in the key pair example), we initialize the object with the private key, then sign a byte array called `data`.

```
PrivateKey priv = pair.getPrivate();
dsa.initSign(priv);
```

```
/* Update and sign the data */
```

```
dsa.update(data);
byte[] sig = dsa.sign();
```

Verifying the signature is straightforward as seen in the code excerpt below:

```
/* Initializing the object with the public key */
PublicKey pub = pair.getPublic();
dsa.initVerify(pub);

/* Update and verify the data */
dsa.update(data);
boolean verifies = dsa.verify(sig);
```

2.5 Digital certificates

Digital certificates introduce a second level to determining the identity of a message originator [15]. To do so, this scheme makes use of certificate authorities, which are third-parties that provide guarantees about signatures and their originators. We examine key and certificate repositories and management tools (keytool and keystore) and discuss the CertPath API, a set of functions designed for building and validating certification paths.

In the digital signature scheme described in the previous section there is clearly a step missing. The mechanism proves that a given party sent a message, does not guarantee that the sender really is who she says she is. For example, what if someone claims to be Alice and signs a message, but is actually somebody else? We can improve our security by using digital certificates, which package an identity along with a public key and are digitally signed by a third party called a certificate authority or CA.

A **certificate authority** is an organization that verifies the identity (in the real-world) of a party and signs that party's public key and identity with the CA private key. A message recipient can obtain the sender's digital certificate and verify (or decrypt) it with the CA's public key. This proves that the certificate is valid and allows the recipient to extract the sender's public key to verify his signature or send him an encrypted message. Browsers and the JDK itself come with built-in certificates and their public keys from several CAs. The JDK supports the X.509 Digital Certificate Standard. For example [14] the following code example reads a file with Base64-encoded certificates. Each call to `generateCertificate` consumes only one certificate, and the read position of the input stream is positioned to the next certificate in the file:

```
FileInputStream fis = new FileInputStream(filename);
BufferedInputStream bis = new BufferedInputStream(fis);

CertificateFactory cf = CertificateFactory.getInstance("X.509");

while (bis.available() > 0) {
    Certificate cert = cf.generateCertificate(bis);
    System.out.println(cert.toString());
}
```

The following example parses a PKCS7-formatted certificate reply stored in a file and extracts all the certificates from it [14]:

```

FileInputStream fis = new FileInputStream(filename);
CertificateFactory cf = CertificateFactory.getInstance("X.509");
Collection c = cf.generateCertificates(fis);
Iterator i = c.iterator();
while (i.hasNext()) {
    Certificate cert = (Certificate)i.next();
    System.out.println(cert);
}

```

2.5.1 Keytool and keystore

The Java platform uses a keystore as a repository for keys and certificates [22]. Physically, the keystore is a file (possibly encrypted) with a default name of keystore. Keys and certificates can have names, called aliases, and each alias can be protected by a unique password. The keystore itself is also protected by a password; you can choose to have each alias password match the master keystore password.

The Java platform uses the keytool to manipulate the keystore. This tool offers many options including generating a public key pair and corresponding certificate, and viewing the result by querying the keystore.

The keytool can be used to export a key into a file, in X.509 format, that can be signed by a certificate authority and then re-imported into the keystore. There is also a special keystore that is used to hold the certificate authority (or any other trusted) certificates, which in turn contains the public keys for verifying the validity of other certificates. This keystore is called the truststore. The Java language comes with a default truststore in a file called cacerts. If one searches for this filename, he will find at least two of these files. One can display the contents with the following command issued from Linux (operating system) command shell:

```

keytool -list -keystore cacerts
Use a password of "changeit"

```

For example, to create a keystore and generate the key pair, you could use a command such as the following [22]:

```

keytool -genkeypair -dname "cn=Mark Jones, ou=JavaSoft, o=Sun, c=US"
    -alias business -keypass kpi135 -keystore /working/mykeystore
    -storepass ab987c -validity 180

```

2.5.2 CertPath API

The Certification Path API is a set of functions for building and validating certification paths or chains. This is done implicitly in protocols like SSL/TLS (see the description in the corresponding section) and JAR file signature verification, but can now be done explicitly in applications with this support.

As mentioned in the section on digital certificates, a CA can sign a certificate with its private key, and if the recipient holds the CA certificate that has the public key needed for signature verification, it can verify the validity of the signed certificate. In this case, the chain of certificates is of length two, the anchor of trust (the CA certificate) and the signed certificate. A self-signed certificate is of length one, the anchor of trust is the signed certificate itself.

Chains can be of arbitrary length, so in a chain of three, a CA anchor of trust

certificate can sign an intermediate certificate; the owner of this certificate can use its private key to sign another certificate. The CertPath API can be used to walk the chain of certificates to verify validity, as well as to construct these chains of trust.

Certificates have expiration dates, but can be compromised before they expire, so Certificate Revocation Lists (CRL) must be checked to really ensure the integrity of a signed certificate. These lists are available on the CA Web sites, and can also be programmatically manipulated with the CertPath API.

3 SECURITY PROTOCOLS

In this section we review the most important existing security protocols that are essential in OpenIoT. We start from the lowest level and follow by higher levels.

3.1 IEEE802.15.4

IEEE802.15.4 is a standard that specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs). IEEE 802.15.4 nodes can operate in either secure mode or non-secure mode. Two security modes are defined in the specification in order to achieve different security objectives: Access Control List (ACL) and Secure mode. The MAC sublayer offers facilities that can be harnessed by upper layers to achieve the desired level of security. Higher-layer processes may specify keys to perform symmetric cryptography to protect the payload and restrict it to a group of devices or just a point-to-point link; these groups of devices can be specified in access control lists. Furthermore, MAC computes *freshness checks* between successive receptions to ensure that presumably old frames, or data that is no longer considered valid, does not transcend to higher layers [14].

In addition to this secure mode, there is another, insecure MAC mode, which allows access control lists merely as a means to decide on the acceptance of frames according to their (presumed) source.

3.2 IPsec

Internet Protocol Security (IPsec) [1] is a technology protocol suite for securing Internet Protocol (IP) communications by authenticating and/or encrypting each IP packet of a communication session. IPsec also includes protocols for establishing mutual authentication between agents at the beginning of the session and negotiation of cryptographic keys to be used during the session.

IPsec is an end-to-end security scheme operating in the Internet Layer of the Internet Protocol Suite. It can be used for protecting data flows between a pair of hosts (*host-to-host*), between a pair of security gateways (*network-to-network*), or between a security gateway and a host (*network-to-host*).

Some other Internet security systems in widespread use, such as Secure Sockets Layer (SSL), Transport Layer Security (TLS) and Secure Shell (SSH), operate in the upper layers of the TCP/IP model. In the past, the use of TLS/SSL had to be designed into an application to protect application protocols. In contrast, since day one, applications did not need to be specifically designed to use IPsec. Hence, IPsec protects any application traffic across an IP network.

The IPsec suite is an open standard and it uses the following protocols to perform various functions:

- ⤴ Authentication Headers (AH) provide connectionless integrity and data origin authentication for IP datagrams and provides protection against replay attacks.
- ⤴ Encapsulating Security Payloads (ESP) provide confidentiality, data-origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic-flow confidentiality.
- ⤴ Security Associations (SA) provide the bundle of algorithms and data that

provide the parameters necessary to operate the AH and/or ESP operations. The Internet Security Association and Key Management Protocol (ISAKMP) provides a framework for authentication and key exchange, with actual authenticated keying material provided either by manual configuration with pre-shared keys, Internet Key Exchange (IKE and IKEv2), Kerberized Internet Negotiation of Keys (KINK), or IPSECKEY DNS records.

Cryptographic algorithms defined for use with IPsec include:

- ⤴ HMAC-SHA1 for integrity protection and authenticity.
- ⤴ TripleDES-CBC for confidentiality
- ⤴ AES-CBC for confidentiality.

3.3 TSL

Transport Layer Security (TLS) [23,2,26] and **Secure Sockets Layer (SSL)** [24], are cryptographic protocols that provide communication security over the Internet, TLS/SSL can be used to authenticate servers and clients and then use it to encrypt messages between the authenticated parties. They use public-key cryptography for authentication of key exchange, private-key encryption for confidentiality and message authentication codes for message integrity. TLS operates in the Application Layer of the IP Protocol Suite. The TLS protocol allows client-server applications to communicate across a network in a way designed to prevent eavesdropping and tampering.

Since protocols can operate either with or without TLS (or SSL) [26], it is necessary for the client to indicate to the server whether it wants to set up a TLS connection or not. There are two main ways of achieving this; one option is to use a different port number for TLS connections (for example port 443 for HTTPS). The other is to use the regular port number and have the client request that the server switch the connection to TLS using a protocol specific mechanism (for example STARTTLS for mail and news protocols).

Once the client and server have decided to use TLS, they negotiate a stateful connection by using a handshaking procedure. During this handshake, the client and server agree on various parameters used to establish the connection's security:

1. The client sends the server the client's SSL version number, cipher settings, session-specific data, and other information that the server needs to communicate with the client using SSL.
2. The server sends the client the server's SSL version number, cipher settings, session-specific data, and other information that the client needs to communicate with the server over SSL. The server also sends its own certificate, and if the client is requesting a server resource that requires client authentication, the server requests the client's certificate.
3. The client uses the information sent by the server to authenticate the server. If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client proceeds to the next step.
4. Using all data generated in the handshake thus far, the client (with the

cooperation of the server, depending on the cipher in use) creates the pre-master secret for the session, encrypts it with the server's public key (obtained from the server's certificate, sent in step 2), and then sends the encrypted pre-master secret to the server.

5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case, the client sends both the signed data and the client's own certificate to the server along with the encrypted pre-master secret.
6. If the server has requested client authentication, the server attempts to authenticate the client. If the client cannot be authenticated, the session ends. If the client can be successfully authenticated, the server uses its private key to decrypt the pre-master secret, and then performs a series of steps (which the client also performs, starting from the same pre-master secret) to generate the master secret.
7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session and to verify its integrity (that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection).
8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished.
9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.

The SSL handshake is now complete and the session begins. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

This is the normal operation condition of the secure channel. At any time, due to internal or external stimulus (either automation or user intervention), either side may renegotiate the connection, in which case, the process repeats itself.

This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the key material until the connection closes. If any one of the above steps fails, the TLS handshake fails and the connection is not created.

In step 3, the client must check a chain of "signatures" from a "root of trust" built into, or added to, the client. The client must *also* check that none of these have been revoked; this is not often implemented correctly, but is a requirement of any public-key authentication system. If the particular signer beginning this server's chain is trusted, and all signatures in the chain remain trusted, then the Certificate (thus the server) is trusted.

Java provides support for TLS/SSL and most Java-based web application frameworks include native support that can be used straight-away by hosted applications. In case of need, the Java API also provides low-level methods to create

a SSL Socket Server and receive messages securely by a client. For example, the following code shows how to create a SSL socket server and listen to messages:

```
SSLServerSocketFactory ssf = (SSLServerSocketFactory) SSLServerSocketF
actory.getDefault();
ServerSocket ss = ssf.createServerSocket(5432);
while (true) {
    Socket s = ss.accept();
    PrintStream out = new PrintStream(s.getOutputStream());
    out.println("Hi");
    out.close();
    s.close();
}
```

3.4 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) [3,17] is a protocol for secure communication over a computer network, widely deployed on Internet. More precisely, it is not a protocol itself but it is rather the result of layering the Hypertext Transfer Protocol (HTTP) on top of SSL/TLS. HTTPS protects against man-in-the-middle attacks by providing authentication of the web site and associated web server that one is communicating with. Additionally, it provides bidirectional encryption of communications between a client and server, which protects against eavesdropping and tampering with and/or forging the contents of the communication. In practice, this provides a reasonable guarantee that one is communicating with precisely the web site that one intended to communicate with (as opposed to an imposter), as well as ensuring that the contents of communications between the user and site cannot be read or forged by any third party.

A site must be completely hosted over HTTPS, without having some of its contents loaded over HTTP, or the user will be vulnerable to some attacks and surveillance. For example, having scripts etc. loaded insecurely on an HTTPS page makes the user vulnerable to attacks. Also having only a certain page that contains sensitive information (such as a log-in page) of a website loaded over HTTPS, while having the rest of the website loaded over plain HTTP will expose the user to attacks. On a site that has sensitive information somewhere on it, every time that site is accessed with HTTP instead of HTTPS, the user and the session will get exposed. Similarly, cookies on a site served through HTTPS have to have the secure attribute enabled.

HTTPS is a URI scheme which has identical syntax to the standard HTTP scheme, aside from its scheme token. However, HTTPS signals the browser to use an added encryption layer of SSL/TLS to protect the traffic. SSL is especially suited for HTTP since it can provide some protection even if only one side of the communication is authenticated. This is the case with HTTP transactions over the Internet, where typically only the server is authenticated (by the client examining the server's certificate).

HTTPS creates a secure channel over an insecure network. This ensures reasonable protection from eavesdroppers and man-in-the-middle attacks, provided that adequate cipher suites are used and that the server certificate is verified and trusted.

Because HTTPS piggybacks HTTP entirely on top of TLS, the entirety of the underlying HTTP protocol can be encrypted. This includes the request URL (which particular web page was requested), query parameters, headers, and cookies (which often contain identity information about the user). However, because host (web site)

addresses and port numbers are necessarily part of the underlying TCP/IP protocols, HTTPS cannot protect their disclosure. In practice this means that even on a correctly configured web server eavesdroppers can still infer the IP address and port number of the web server (sometimes even the domain name e.g. `www.example.org`, but not the rest of the URL) that one is communicating with as well as the amount (data transferred) and duration (length of session) of the communication, though not the content of the communication.

Web browsers know how to trust HTTPS websites based on certificate authorities that come pre-installed in their software. Certificate authorities (e.g. VeriSign/Microsoft/etc.) are in this way being trusted by web browser creators to provide valid certificates. Logically, it follows that a user should trust an HTTPS connection to a website if and only if all of the following are true:

- ⤴ The user trusts that the browser software correctly implements HTTPS with correctly pre-installed certificate authorities.
- ⤴ The user trusts the certificate authority to vouch only for legitimate websites.
- ⤴ The website provides a valid certificate, which means it was signed by a trusted authority.
- ⤴ The certificate correctly identifies the website (e.g., when the browser visits "`https://example.com`", the received certificate is proper for "Example Inc." and not some other entity).
- ⤴ Either the intervening hops on the Internet are trustworthy, or the user trusts that the protocol's encryption layer (TLS/SSL) is sufficiently secure against eavesdroppers.

HTTPS is especially important over unencrypted networks (such as WiFi), as anyone on the same local network can "packet sniff" and discover sensitive information. Additionally, many free to use and even paid for WLAN networks do packet injection for serving their own ads on webpages or just for pranks, however this can be exploited maliciously e.g. by injecting malware and spying on users.

Another example where HTTPS is important is connections over Tor (anonymity network), as malicious Tor nodes can damage or alter the contents passing through them in an insecure fashion and inject malware into the connection. This is one reason why the Electronic Frontier Foundation and Torproject started the development of HTTPS Everywhere, which is included in the Tor Browser Bundle.

In most Java-based web application frameworks, SSL and HTTPS are provided via configuration, and are orthogonal to the application itself. As an example, in JBoss (where the OpenIoT platform demonstrator has been tested) this can be achieved following the steps documented in the official web site¹.

¹ <https://docs.jboss.org/jbossweb/3.0.x/ssl-howto.html>

4 ACCESS CONTROL

In this section we review the fundamental concepts of access control [28,27,18], including the fundamental concepts of identities, authentication and authorisation. Access control mechanisms are security features that manage the way how users and systems communicate with other systems and resources, respecting policies of information flow, as we will see.

4.1 Fundamental concepts

Access control is a selective restriction of access to resource and includes the following components: authentication, authorisation, access approval and accountability. The *Access* can be interpreted as the flow of information between a subject (i.e. an active entity requesting access) and an object (a passive entity that contain information to be accessed). Subjects and objects should both be considered as software entities, rather than as human users: any human user can only have an effect on the system via the software entities that they control.

Access controls give organization the ability to control, restrict, monitor, and protect resource availability, integrity and confidentiality. Access control consists of the different sub-processes detailed below [27].

- **Identification** is a method of ensuring that a subject is the entity it claims to be, e.g. assign an identity to a subject.
- **Authentication** is the process of verifying that an identity is bound to the entity that makes an assertion or claim of identity (i.e., verifying that "you are who you say you are"). Authenticators are commonly based on "*something you know*", such as a password or a personal identification number (PIN). This assumes that only the owner of the account knows the password or PIN needed to access the account.
- **Authorisation** is the act of defining access rights for subjects. An authorisation policy specifies the operations that subjects are allowed to execute on the system.
- **Access approval** is the function that actually grants or rejects access during operations. During access approval the system compares the formal representation of the authorisation policy with the access request to determine whether the request shall be granted or rejected.
- **Accountability** uses such system components as *audit trails* (records) and *logs* to associate a subject with its actions. The information recorded should be sufficient to map the subject to a controlling user. Audit trails and logs are important for detecting security violations and re-creating security incidents

Two most widely recognized access control models are **Discretionary Access Control (DAC)** and **Mandatory Access Control (MAC)**.

- **Discretionary Access Control (DAC)** [30] is a policy determined by the owner of an object. The owner decides who is allowed to access the object and what privileges they have. Two important concepts in DAC are:
 - File and data ownership: Every object in the system has an *owner*. In most DAC systems, each object's initial owner is the subject that

caused it to be created. The access policy for an object is determined by its owner.

- Access rights and permissions: These are the controls that an owner can assign to other subjects for specific resources.
- **Mandatory Access Control (MAC)** [31] *refers to allowing access to a resource if and only if rules exist that allow a given user to access the resource.* It is difficult to manage but its use is usually justified when used to protect highly sensitive information. Examples include certain government and military information. Management of MAC is often implemented by using **sensitivity labels**. In such a system subjects and objects must have labels assigned to them. A subject's sensitivity label specifies its level of trust. An object's sensitivity label specifies the level of trust required for access. *In order to access a given object, the subject must have a sensitivity level equal to or higher than the requested object.*

Two methods are commonly used for implementing mandatory access control using sensitivity labels:

- **Rule-based access control:** This type of control further defines specific conditions for access to a requested object. A Mandatory Access Control system implements a simple form of rule-based access control to determine whether access should be granted or denied by matching:
 1. An object's sensitivity label
 2. A subject's sensitivity label
- **Lattice-based access control:** A lattice is used to define the levels of security that an object may have and that a subject may have access to. The subject is only allowed to access an object if the security level of the subject is greater than or equal to that of the object. A lattice model is a mathematical structure that defines greatest lower-bound (meet) and least upper-bound (join) values for a pair of elements, such as a subject and an object. For example, if two subjects *A* and *B* need access to an object, the security level is defined as the meet of the levels of *A* and *B*. In another example, if two objects *X* and *Y* are combined, they form another object *Z*, which is assigned the security level formed by the join of the levels of *X* and *Y*.

Access Control Matrix or **Access Matrix** is an abstract, formal security of protection state in computer systems, that characterize the rights of each subject with respect to every object in the system. More formally, access matrix is defined as a set of objects *O*, that is the set of entities that needs to be protected (e.g., processes, files, memory pages) and a set of subjects *S*, that consists of all active entities (e.g., users, processes). Further there exists a set of rights *R* of the form $r(s, o)$, where $s \in S$, $o \in O$ and $r(s, o) \in R$, where a right specifies the kind of access a subject is allowed to process object. Consider the following example of a matrix where there exist two subjects (*Role1* and *Role2*) the following objects: *asset1*, *asset2*, *file* and a *device*. Table 1 presents an example access matrix.

Table 1. An example access matrix

	<i>asset 1</i>	<i>asset 2</i>	<i>file</i>	<i>device</i>

<i>Role1</i>	<i>read, write, execute, own</i>	<i>execute</i>	<i>read</i>	<i>write</i>
<i>Role2</i>	<i>Read</i>	<i>read, write, execute, own</i>		

In general, all subjects are objects but the inverse is not true. Thus, the cell for row s and column o , $[s, o]$, denotes the set of rights of subject s to perform an operation on object o (e.g., *read in* $[s, o]$). Thus, all users in the access matrix are represented by their corresponding subjects. The access matrix is a dynamic entity and its individual cells can be modified by subjects. For example, if subject s is the owner of object o then s can modify the content of cells corresponding to o . In such a case the owner of the object has complete discretion regarding the access to the owned object by other subjects. Such an access control model is called discretionary. The access matrix is usually sparse and is stored in a system using access control lists, capabilities, relations or another data structure suitable for efficient sparse matrix storage.

An **access control list (ACL)**, with respect to a computer system, is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects. Each entry in a typical ACL specifies a subject and an operation. For instance, if a *file* has an ACL that contains (*Alice, delete*), this would give *Alice* permission to delete the *file*.

4.2 Lattice-Based Access Control Models

In this section we review foundations of lattice-based access control models given the importance of lattice-based access control systems [6].

A security of computer system has the following interdependent objectives:

- ✧ **Confidentiality** (or secrecy) related to disclosure of information, i.e., preventing users from learning about data of other users.
- ✧ **Integrity**, related to modification of information, i.e., preventing a user from changing data of other users.
- ✧ **Availability** related to denial of access to information, i.e., ensuring that requested data is delivered on time.
- ✧ **Non-repudiation** related to providing a proof of the integrity and origin of data.

Lattice-based access control models were developed to deal with information flow in computer system. Although developed for the defence sector they can be used in most cases where information flow is critical. Therefore they are a key component of computer security.

4.2.1 Information flow policy

Information flow policies are concerned with the flow of information from one security class to another. In particular in a computer system this information flows from one object to another, where object is defined as a container of information (e.g., files and directories in an operating system).

Information flow is controlled by assigning every object a security class also called a security label. If information flows from object x to object y , it implies information flow from the security class of x to the security class of y . Thus, an information flow from one class to another, concerns the corresponding objects.

An information flow policy can be formally defined as follows:

- SC is the set of security classes
- $A \rightarrow B$ is a binary relation on SC called “flows”, where A, B is in SC . Thus, the information flows from security class A to security class B .
- $A + B = C$ is a binary class combination or join operation, where C is in SC . Thus, the join operation specifies how to label information obtained by combining information from two security classes A and B , where C is the resulting class.
- ' \geq ' is the dominance operator, where $A \geq B$ (A dominates B) if and only if $B \rightarrow A$. The strict dominates relation $>$ is defined by $A > B$ if $A \geq B$ and $A \neq B$. Thus, if $A > B$ then $B \rightarrow A$.
Then the definition of $A + B$ is just the maximum with respect to A and B . Thus, if information from two security classes is combined the label of the higher of the two is used for the result.

As an example consider High-low policy defined as follows: $SC = \{H, L\}$, the flow relations is $H \rightarrow H, L \rightarrow L, L \rightarrow H$ and the join is defined as follows: $H + H = H, L + H = H, L + L = L$.

As another example of security classes consider $SC = \{TS: \text{top secret}, S: \text{secret}, C: \text{confidential}, U: \text{unclassified}\}$ and the total (linear) ordering of the security classes as follows: $U \rightarrow S \rightarrow C \rightarrow TS$ meaning $TS > S > C > U$ and $A + B$ is the maximum with respect to the dominance relation.

4.3 Access control models-based on information flow

Recall, that a user is defined as a human being assigned a unique user *id* in the system. A subject is a process in the system (a program in execution), where each subject is associated with a single user. In general a user can have several subjects concurrently running in the system. Thus, every time a user logs into the system it does so as a particular subject. (Note that access control models assume that identification and authentication of users takes place in a secure and correct manner. Different subjects associated with the same user can obtain different sets of access rights. For example, top secret user Bob logs in at the secret level. Then Bob can have subjects running at different levels dominated by the top secret class.

The discretionary access control model is not adequate for enforcing information flow policies because they provide no constraints on copying information from one object to another. For an illustration of this property consider the following example.

EXAMPLE 1. Suppose that *Tom*, *Dick* and *Harry* are users and *Tom* has a confidential file *Private* that he wants *Dick* to read but does not want *Harry* to read. *Tom* can authorize *Dick* to read the file by entering *read(Dick, Private)* in the access matrix. *Dick* can easily subvert *Tom*'s intention by creating a new file called *Copy-of-Private* and copying the contents of *Private* into it. As the creator of *Copy-of-Private*, *Dick* has the authority to grant read access for it to any user including *Harry*. Thus, *Dick* can enter *read (Harry,Copy-of-Private)* in the access matrix. Then *Harry* can

read *Private*.

4.3.1 Mandatory access policy

The key idea of the mandatory access control model that we present in this section is *to augment discretionary access controls with mandatory access controls to enforce information flow policies*.

Thus, we take a two-step approach to access control. First a discretionary access matrix D , whose contents can be modified by subjects, is used, where authorisation in D is not sufficient for an operation to be carried out. Second, the operation must be authorized by the mandatory access control policy over which users have no control.

4.3.2 Confidentiality

Mandatory access control policy is expressed in terms of security labels attached to subjects (**security classification**) and objects (**security clearance**). Thus, a user labelled *secrete* can run the same program such as text editor, as a subject labelled *secret* or as a subject labelled *unclassified*. Even though both subjects run the same program on behalf of the same user they both obtain different privileges due to their security labels. The assumption called tranquillity says that the security labels on subjects and objects cannot be changed.

Let L be the security label (confidentiality) of a given subject or object. Then mandatory access rules can be expressed as follows:

- Subject s can read an object o only if $L(s) \geq L(o)$, meaning $L(o) \rightarrow L(s)$, i.e., o flows to s .
- Subject s can write object o only if $L(s) \leq L(o)$, meaning $L(s) \rightarrow L(o)$, i.e., s flows to o .

The mandatory checks are only applied if the checks of the discretionary matrix have been satisfied. If the matrix does not authorize the operation then we do not check the mandatory controls.

These security requirements apply to humans and programs equally. The write property is not applied to humans but to programs. Human users are trusted not to leak information. A secret user can write unclassified document because we assume that he will put only unclassified information in it.

Programs are not trusted because they can have embedded Trojan Horses. The write property prohibits a program running at a *secrete* level from writing to unclassified objects even if it is permitted to do so by discretionary access control. A user labelled *secret* can write to an unclassified object must log as an unclassified subject.

Now consider how the presented properties impact Example 1. Clearly the *read* property will prevent *Harry's* subjects from being able to directly read the file *Private*. The presented *write* property will ensure that *Harry's* subjects cannot surreptitiously read *Copy-of-Private* because it will either be labelled *secret* or will not contain any information from *Private*.

4.3.3 Integrity

The presented model can be extended to handle integrity. The concept of integrity says that low-integrity information should not be allowed to flow to high-integrity

objects. Let W denote the integrity label of a subject or object. A particular mandatory integrity rules can be expressed as follows:

- Subject s can read an object o only if $W(s) \leq W(o)$, meaning $W(s) \rightarrow W(o)$, i.e., s flows to o .
- Subject s can write an object o only if $W(s) \geq W(o)$, meaning $W(o) \rightarrow W(s)$, i.e., o flows to s .

4.3.4 Combined

It is often suggested that the confidentiality and integrity models could be combined in situations where both confidentiality and integrity are of concern. If a single label were used for both, confidentiality and integrity then a model would impose conflicting constraints. Therefore, a model with independent confidentiality and integrity labels is more useful in practice. In such a model each security class consists of two labels: a confidentiality label L and an integrity label W with independent controls applied to them. We assume that that in both lattices high confidentiality and high integrity are at the top.

Example combined confidentiality and integrity mandatory rules can be expressed as follows:

- Subject s can read an object o only if $L(s) \Rightarrow L(o)$ and $W(s) \leq W(o)$
- Subject s can write an object o only if $L(s) \leq L(o)$ and $W(s) \geq W(o)$

This popular combined model has been implemented in several operating system, database and network products specifically built to meet requirements of the military sector. Thus, this model amounts to the simultaneous application of two lattices, with information flow, occurring in opposite directions (going upward in the confidentiality lattice and downward in the integrity lattice).

5 THE SECURITY ARCHITECTURE IN OPENIOT

In this section we present the proposed security architecture in OpenIoT that accommodates the specifics/requirements of the OpenIoT platform.

5.1 Overview of the architecture

Clearly, the OpenIoT platform consist of several cooperating distributed standalone applications (e.g., SDUM, X-GSN, LSM, etc.) that require individual security (authentication and authorisation) services because of differing subjects and objects that they deal with. Therefore, we propose a central authorisation and authentication server that provides authentication and authorisation services for all relevant OpenIoT applications running on behalf of different subjects.

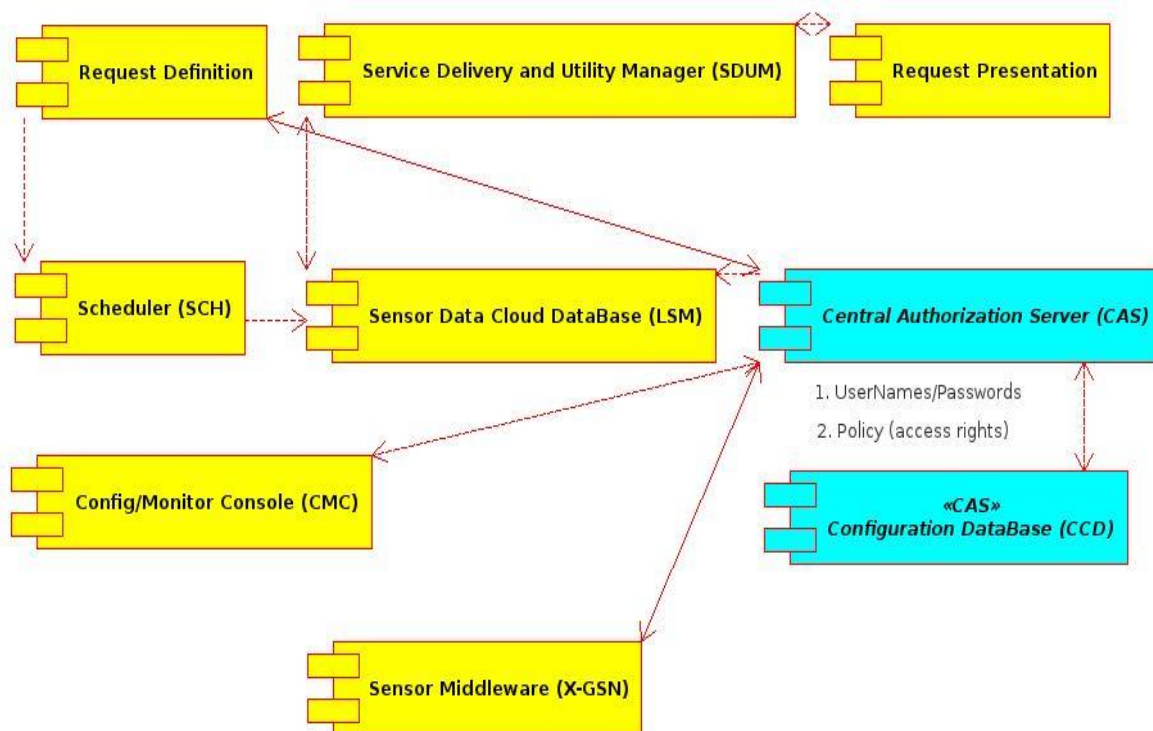


Figure 1. Security Architecture in OpenIoT.

The main feature of this architecture is that user credentials (username/password) are only checked and maintained by Central Authorisation Server (CAS) and it authorizes applications running on behalf a user by granting them an access token with a given time to live. This prevents any circulation of the credentials throughout OpenIoT components. Furthermore, the architecture has to be open to external security schemes that are more flexible or secure (or both) than the default security model. Note that the security and performance are usually orthogonal features and strong security means lower performance and vice-versa. Therefore, our ultimate objective is to design a scheme where security can be traded for performance.

Figure 1 presents an overview of the security architecture in the OpenIoT architecture.

In particular, CAS architecture considers the following roles:

- ⌘ Resource owner: e.g., an owner of sensor data in LSM that grants access to

the data for Service Delivery and Utility Manager (SDUM).

- ✧ Resource server: e.g., Sensor Data Cloud Database (LSM).
- ✧ Client: e.g., SDUM querying LSM on behalf of a resource owner. In general, Client consists of the distributed set of OpenIoT components that use the corresponding token to get access to the data of the corresponding Resource owner in LSM.
- ✧ Authorisation server: CAS issuing access tokens to a client after successfully authenticating the resource owner and obtaining authorisation.

The Clients that directly authenticate with CAS are:

- ✧ Request Definition: where a user defines a request (query) and upon authentication the other clients (e.g., SCH, LSM, Request Presentation) get a corresponding token to accomplish their tasks.
- ✧ CMC: where the administrator is authenticated to accomplish his tasks.
- ✧ X-GSN: where sensor data providers are authenticated to stream the corresponding data to LSM.

The details of the CAS architecture are presented in Section 6.

5.2 Access matrix

The utilized access matrix consists of the following subjects and objects. The roles are as follows:

- ✧ Administrator: This role gains access to the entire OpenIoT platform. All the actions of the different modules will be available to administrators. Administrators will also have access to all available GUIs.
- ✧ User: The user role is the most common and the most used role of the access control model. After creating an account, the user will gain access to a specific list of possible actions. The user will have access to the request definition and the request presentation GUIs.

The considered objects are as follows:

1. Physical sensor (PS).
2. Virtual sensors (VS).
3. Internet Connected Object (ICO).
4. Services related to the components (e.g., SDUM, SCH and LSM operations).

A fragment of the access matrix as a relation is presented in Table 2.

Table 2. A fragment of the access matrix in OpenIoT.

Role	Relation r(o)
Administrator	setup(VS), setup(PS), setup(LSM), setup(SCH), setup(SDUM), write(LSM), read(LSM), ..., etc.
User	read(VS), read(PS), read(GUI), ..., etc.

5.3 Mandatory access rules

We consider the following security classes

1. TS: top secret
2. C: confidential.

Security classes assignment to roles is presented in Table 3.

Table 3. Security classes and roles in OpenIoT.

Role	Security class (confidentiality label)
Administrator	TS: top secret
User	C: confidential

We adapt the confidentiality rules as trust rules from Section 4. 3.

5.4 Security protocol stack

The security protocol stack in OpenIoT is presented in Table 4, where

The layers refer to corresponding communication between the following components: Layer 1: sensors and GSN, Layer 2: GSN and LSM and Layer 3: other OpenIoT components (e.g., LSM and SDUM).

Table 4. The security protocol stack in OpenIoT.

	Layer/Link	Security/privacy solution
1.	Sensor->GSN	Ipssec (wired networks), IEEE802.15.4 (wireless networks)
2.	GSN->LSM	TSL, HTTPS
3.	Application (OpenIoT)	TSL, HTTPS

Clearly, the OpenIoT platform relies entirely on the TSL/HTTPS protocol to ensure secure (encrypted) messaging, while IEEE802.15.4, Ipssec guarantees secure sensor data.

6 AUTHORISATION FRAMEWORK

In this section we review the OAuth [4, 5, 12] authorisation framework that is used to provide authorisation to OpenIoT modules on behalf of a user.

We chose OAuth as our authorisation framework for the following reasons:

- ✧ From the point of view of OpenIoT the most important fact about OAuth is that it describes a method for providing authorisation in a distributed environment, where distributed client applications get access to owner's resources using time-stamped tokens to avoid transmitting credentials (username, password) to the client applications.
- ✧ OAuth is an open standard for authorisation, i.e., publicly available, and developed, approved and maintained via a collaborative and consensus driven process.
- ✧ OAuth is more like a framework (not a defined protocol), which leaves a lot of implementation freedom that we need because of non-standard requirements in OpenIoT. For example, for some OpenIoT applications involving mobile phones as sensors it is important to have time stamped and location-restricted access tokens.
- ✧ OAuth is the only framework in its genre and is widely used for similar applications.

Furthermore, OAuth is a result of standardization and combined wisdom of many well-established industry protocols. It is similar to other protocols currently in use (Google AuthSub, AOL OpenAuth, Yahoo BBAuth, Flickr API, Amazon Web Services API, etc). Each protocol provides a proprietary method for exchanging user credentials for an access token or ticker. OAuth was created by carefully studying each of these protocols and extracting the best practices and commonality that allow new implementations as well as a smooth transition for existing services to support OAuth [12].

An area, where OAuth is more evolved than some of the other protocols and services is its direct handling of non-website services. OAuth was built in support for desktop applications, mobile devices, set-top boxes, and of course websites. Many of the protocols today use a shared secret hardcoded into software but such an approach may pose an issue when the service trying to access private data is open source [12]. In the following part of Section 6 the most important excerpts from the OAuth 2.0 framework specification [5] are introduced and discussed.

6.1 Introduction

In the traditional client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party.

This creates several problems and limitations:

- ✧ Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.

- ⚠ Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- ⚠ Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- ⚠ Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
- ⚠ Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorisation layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token a string denoting a specific scope, lifetime, and other access attributes. An authorisation server with the approval of the resource owner issues access tokens to third-party clients. The client uses the access token to access the protected resources hosted by the resource server.

For example, an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo-sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo-sharing service (authorisation server), which issues the printing service delegation-specific credentials (access token).

6.2 Roles

OAuth defines four roles:

1. **Resource owner:** an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
2. **Resource server:** the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
3. **Client:** an application making protected resource requests on behalf of the resource owner and with its authorisation. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
4. **Authorisation server:** the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorisation.

The interaction between the authorisation server and resource server is beyond the scope of this specification. The authorisation server may be the same server as the resource server or a separate entity. A single authorisation server may issue access tokens accepted by multiple resource servers.

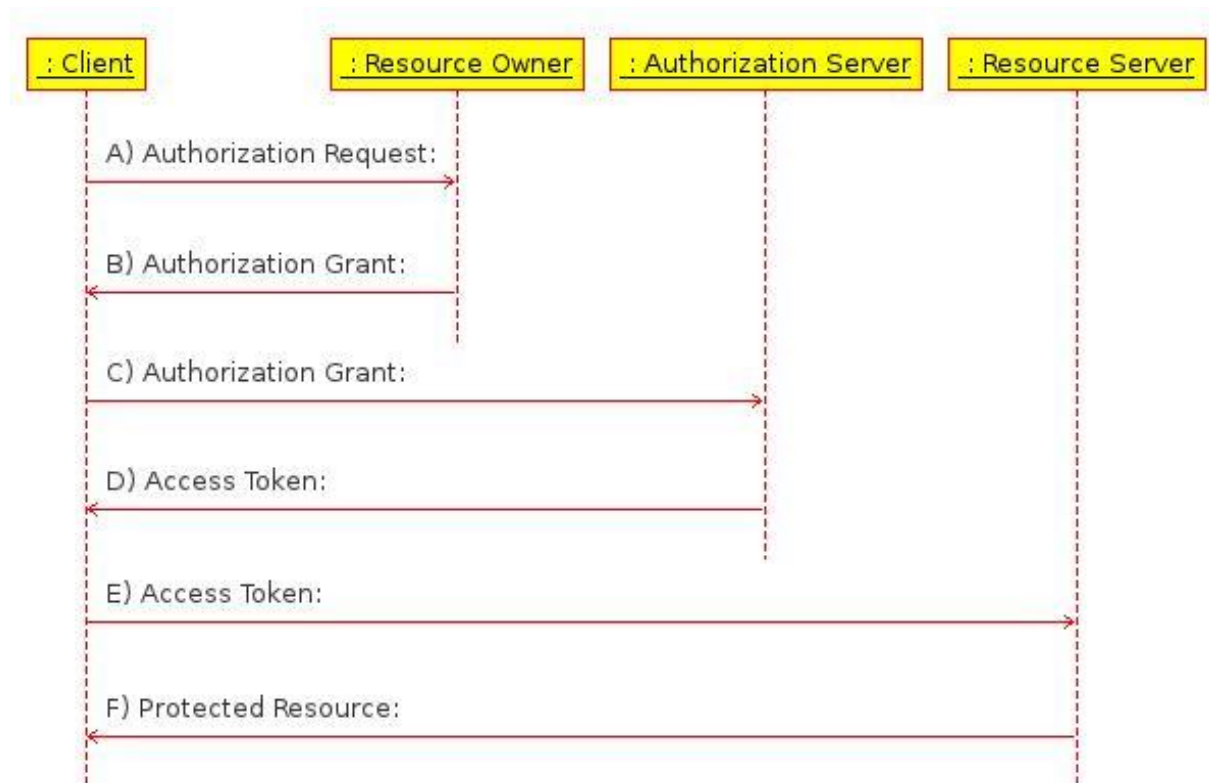


Figure 2. Abstract Protocol Flow.

6.3 OAuth Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 2 describes the interaction between the four roles and includes the following steps:

- A) The client requests authorisation from the resource owner. The authorisation request can be made directly to the resource owner (as shown), or preferably indirectly via the authorisation server as an intermediary.
- B) The client receives an authorisation grant, which is a credential representing the resource owner's authorisation, expressed using one of four grant types defined in this specification or using an extension grant type. The authorisation grant type depends on the method used by the client to request authorisation and the types supported by the authorisation server.
- C) The client requests an access token by authenticating with the authorisation server and presenting the authorisation grant.
- D) The authorisation server authenticates the client and validates the authorisation grant, and if valid, issues an access token.
- E) The client requests the protected resource from the resource server and authenticates by presenting the access token.
- F) The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorisation grant from the resource owner (depicted in steps (A) and (B)) is to use the authorisation server as an

intermediary as in Figure 3.

6.3.1 Authorisation Grant

An authorisation grant is a credential representing the resource owner's authorisation (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types: authorisation code, implicit, resource owner password credentials, and client credentials as well as an extensibility mechanism for defining additional types.

6.3.2 Authorisation Code

The authorisation code is obtained by using an authorisation server as an intermediary between the client and resource owner. Instead of requesting authorisation directly from the resource owner, the client directs the resource owner to an authorisation server, which in turn directs the resource owner back to the client with the authorisation code.

Before directing the resource owner back to the client with the authorisation code, the authorisation server authenticates the resource owner and obtains authorisation. Because the resource owner only authenticates with the authorisation server, the resource owner's credentials are never shared with the client.

The authorisation code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others, including the resource owner.

The authorisation code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorisation server.

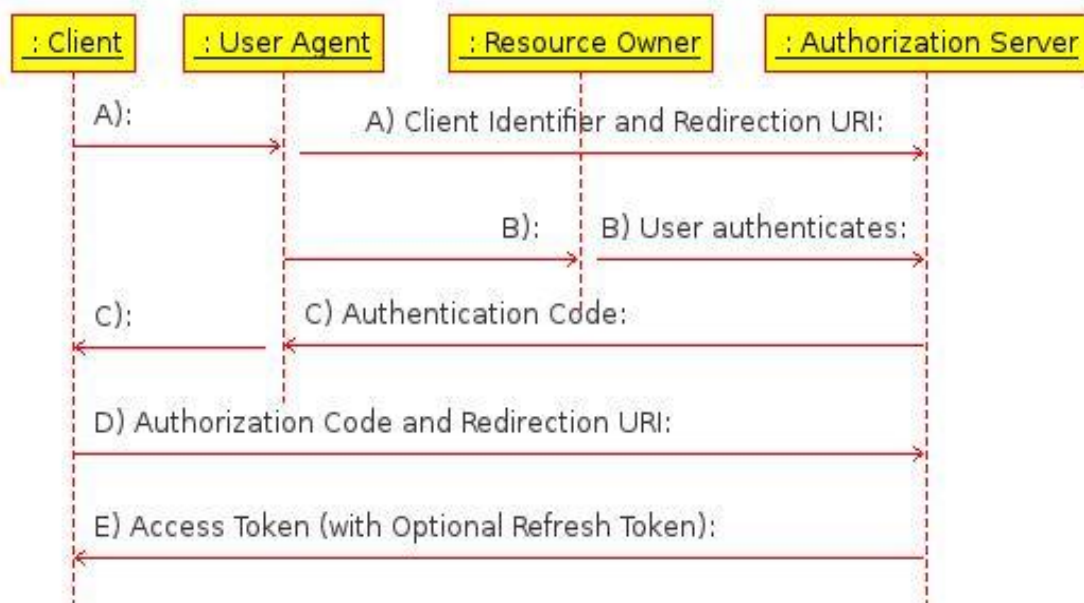


Figure 3. Authorisation Code Flow.

The flow illustrated in Figure 3 includes the following steps:

- A) The client initiates the flow by directing the resource owner's user-agent to the authorisation endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorisation server will send the user-agent back once access is granted (or denied).
- B) The authorisation server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.
- C) Assuming the resource owner grants access, the authorisation server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorisation code and any local state provided by the client earlier.
- D) The client requests an access token from the authorisation server's token endpoint by including the authorisation code received in the previous step. When making the request, the client authenticates with the authorisation server. The client includes the redirection URI used to obtain the authorisation code for verification.
- E) The authorisation server authenticates the client, validates the authorisation code, and ensures that the redirection URI received matches the URI used to redirect the client in step C). If valid, the authorisation server responds back with an access token and, optionally, a refresh token.

6.3.3 Implicit

The implicit grant is a simplified authorisation code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorisation code, the client is issued an access token directly (as the result of the resource owner authorisation). The grant type is implicit, as no intermediate credentials (such as an authorisation code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorisation server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application), since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants, especially when the authorisation code grant type is available.

6.3.4 Resource Owner Password Credentials

The resource owner password credentials (i.e., username and password) can be used directly as an authorisation grant to obtain an access token. The credentials

should only be used when there is a high degree of trust between the resource owner and the client (e.g., the client is part of the device operating system or a highly privileged application), and when other authorisation grant types are not available (such as an authorisation code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

6.3.5 Client Credentials

The client credentials (or other forms of client authentication) can be used as an authorisation grant when the authorisation scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorisation server. Client credentials are used as an authorisation grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorisation previously arranged with the authorisation server.

6.4 Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorisation issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorisation server.

The token may denote an identifier used to retrieve the authorisation information or may self-contain the authorisation information in a verifiable manner (i.e., a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorisation constructs (e.g., username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorisation grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g., cryptographic properties) based on the resource server security requirements.

6.5 Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorisation server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorisation server. If the authorisation server issues a refresh token, it is included when issuing an access token (i.e., step (D) in Figure 2).

A refresh token is a string representing the authorisation granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorisation information. Unlike access tokens, refresh tokens are intended for use only with authorisation servers and are never sent to resource servers.

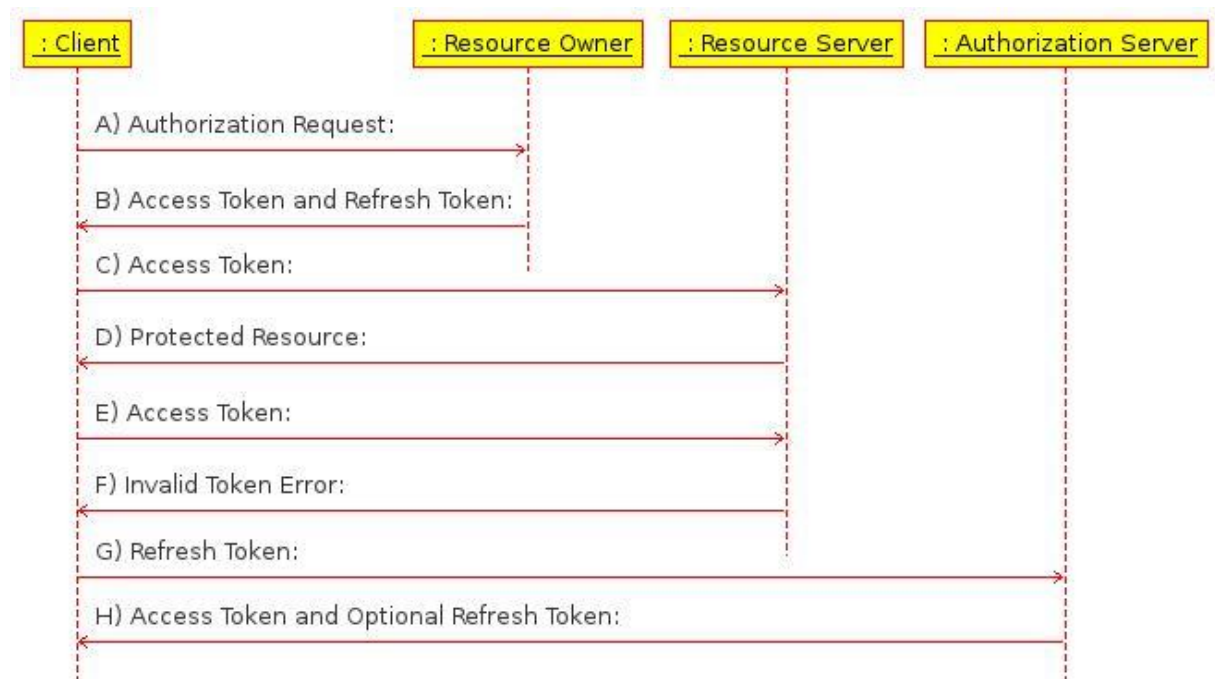


Figure 4. Refreshing an Expired Access Token.

The flow illustrated in Figure 4 includes the following steps:

- A) The client requests an access token by authenticating with the authorisation server and presenting an authorisation grant.
- B) The authorisation server authenticates the client and validates the authorisation grant, and if valid, issues an access token and a refresh token.
- C) The client makes a protected resource request to the resource server by presenting the access token.
- D) The resource server validates the access token, and if valid, serves the request.
- E) Steps C) and D) repeat until the access token expires. If the client knows the access token expired, it skips to step G); otherwise, it makes another protected resource request.
- F) Since the access token is invalid, the resource server returns an invalid token error.
- G) The client requests a new access token by authenticating with the authorisation server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorisation server policies.

- H) The authorisation server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).

6.6 TLS Version

Whenever Transport Layer Security (TLS) is used by this specification, the appropriate version (or versions) of TLS will vary over time, based on the widespread deployment and known security vulnerabilities.

Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

6.7 HTTP Redirections

This specification makes extensive use of HTTP redirections, in which the client or the authorisation server directs the resource owner's user-agent to another destination. While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

6.8 Interoperability

OAuth 2.0 provides a rich authorisation framework with well-defined security properties. However, as a rich and highly extensible framework with many optional components, on its own, this specification is likely to produce a wide range of non-interoperable implementations.

In addition, this specification leaves a few required components partially or fully undefined (e.g., client registration, authorisation server capabilities, endpoint discovery). Without these components, clients must be manually and specifically configured against a specific authorisation server and resource server in order to interoperate.

This framework was designed with the clear expectation that future work will define prescriptive profiles and extensions necessary to achieve full web-scale interoperability.

6.9 Security considerations

For the security considerations, we refer the reader to the RFC <http://tools.ietf.org/html/rfc6819>, which describes the security framework of OAuth 2.0.

We studied OAuth [12] in addition to the framework recommendations RFC2818 [4], RFC5849 [5] because in [12] the details how the OAuth framework achieves high standards in terms of security are more detailed and illustrated. OAuth security specification is considered to OpenIoT implementation providing a good starting point for a scalable security implementation. The OAuth security guide [12] is written by the lead author of the RFC2818, RFC5849 therefore, we consider the documents as complementary and necessary in order to obtain the whole picture of the OAuth framework, from the first hand, that is very important not only for developers of OpenIoT but also for general audience to demonstrate that OAuth is based on proven security mechanisms and protocols as it is cited in this project deliverable.

7 TRUSTWORTHINESS OF SENSOR READINGS

In this section we describe an approach for trust-management in an IoT environment, specifically targeting sensor data readings. When sensor data is acquired from large and heterogeneous crowds, the trustworthiness of each sensor reading can be compromised, consequently lowering the overall utility of the system. In this context, it is important for an IoT platform to be able to assess how much a sensor data observation can be trusted.

The main outcome of this work is an **algorithm** that precisely provides an assessment of sensor data trustworthiness, based on the principle of comparing data produced by similar (sometimes overlapping or redundant) sensors. Spatial correlation has a meaning in the context of sensor data in a variety of monitoring applications, where a key characteristic is that nearby sensor nodes monitoring an environmental feature typically register similar values [9, 10]. This kind of data redundancy due to the spatial correlation between sensor observations is the corner-stone of the proposed algorithm for assessing trustworthiness of sensor readings. Thus, we use the spatial neighbourhood of a given sensor to compare its values with values of the neighbourhood to assess trustworthiness of the sensor.

The most related to our work is [7, 8], where a provenance and a game-theoretic approach for assessing trustworthiness in sensor networks were proposed. However, they did not consider spatio-temporal correlations between the sensor streams. In [11] a multi-stream join was proposed for mining spatio-temporal correlations between multiple streams.

7.1 Notation

We use superscript (i) to refer to the i-th stream. $A^{(i)} = \{a_1^{(i)}, a_2^{(i)}, \dots, a_{m(i)}^{(i)}\}$ is an alphabet in stream i. $S = \{s^{(1)}, s^{(2)}, \dots, s^{(i)}\}$ is a multi-stream defined as a set of input streams each of a possibly different length $n^{(i)}$ resulting from a different rate of generating symbols $s^{(i)} = [s_1^{(i)}, s_2^{(i)}, \dots, s_{n(i)}^{(i)}]$ is the i-th stream (i-th attribute sequence). Every stream tuple (stream element) has three attributes: (I) timestamp $s_t^{(i)}.timestamp = t$, where $t \in \{1, 2, \dots\}$; (II) stream identifier $s_t^{(i)}.stream = i$ and (III) (relational tuple) denoted $s_t^{(i)}.value$, where $s_t^{(i)}.value \in A^{(i)}$. For simplicity we just use $s_t^{(i)}$ to refer to $s_t^{(i)}.value$. $X_t^{(k)}$ is the random variable corresponding to the value of the data point of sensor k at time t. $N_t^{(k)}$ be the set of neighbours (neighbourhood) of sensor k at time t. $E(N_t^{(k)})$ and $Var(N_t^{(k)})$ are the average and the variance of the data points of the neighbours at time t. $T^{(k)} = [T_{ts}^{(k)}, T_{ts+1}^{(k)}, \dots, T_{te}^{(k)}]$ is the trust stream of the k-th sensor for the time window $[ts, te]$, where ts is the start time and te is the end time. $T_t^{(k)}$ is the trust score for sensor k at time t, where $T_t^{(k)} \in [0, 1]$.

7.2 Problem definition

The problem of computing trustworthiness of sensors readings based on spatio-temporal correlation with neighbours can be defined as follows:

Given:

- an input collection of spatio-temporally correlated (temporary overlapping neighbouring streams) streams $S = \{s^{(1)}, s^{(2)}, \dots, s^{(|I|)}\}$, where $s^{(i)} = [s_{t(i)s}^{(i)}, s_{t(i)s+1}^{(i)}, \dots, s_{t(i)e}^{(i)}]$ of possibly of different lengths.
- $x^{(i)}$ and $y^{(i)}$ are the coordinates of sensor stream $s^{(i)}$.
- $t(i)s$ and $t(i)e$ are the start and end timestamp of sensor stream $s^{(i)}$ respectively.
- sensor identifier k .

Task: compute the trust sequence $T^{(k)} = [T_{ts}^{(k)}, T_{ts+1}^{(k)}, \dots, T_{te}^{(k)}]$.

7.3 Overview of the method

Consider sensor streams $s^{(1)}, s^{(2)}, s^{(3)}, s^{(4)}$ in Figure 5, and the task of assessing a corresponding trust stream for sensor $s^{(1)}$ called $T^{(1)}$ based on its spatio-temporal correlation with neighbouring streams. Thus, the general idea of our approach, to computing a trust stream for a given sensor node, is to compare its values with values of its neighbouring sensors. Then the more the streams of the neighbours are similar the higher the trust of the sensor. Note the neighbouring streams may not completely cover the time span of $s^{(1)}$ (presence of discontinuities) that makes the problem more difficult than simply correlating streams.

Thus, our method works as follows: (I) we assess the spatio-temporal correlations between the streams to form the neighbourhood for each sensor in a time period where we assume all streams are trustworthy and (II) given the neighbourhood we assess the trustworthiness of the sensors by comparing them to the centroids of the neighbourhood.

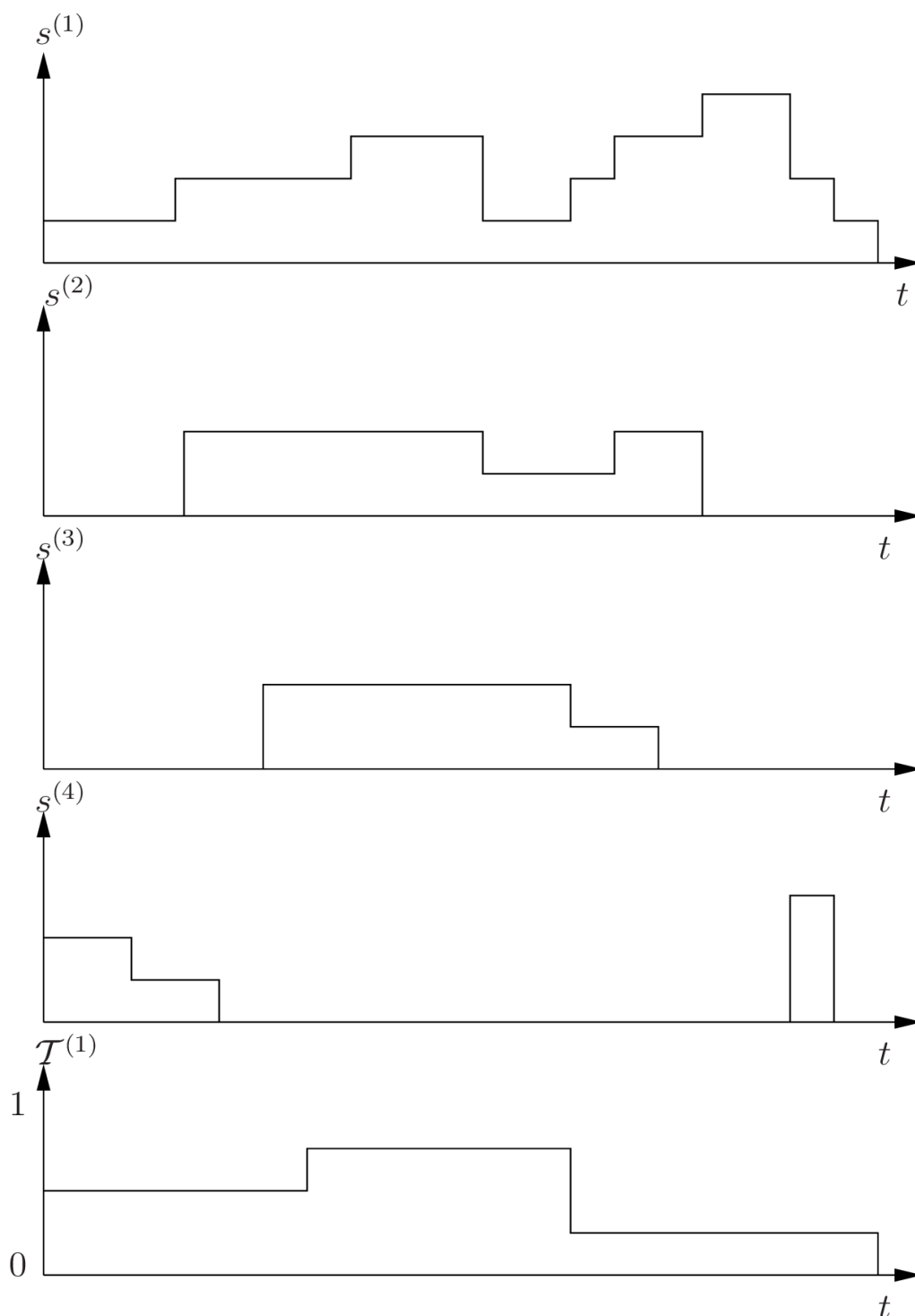


Figure 5. Trust of Sensor Stream Representations.

7.4 Algorithm

We associate a trust score with each data point that provides an indication about trustworthiness of the data point. The more trustworthy data a source provides the more trusted is the source. Thus, there is an interdependence between trust

scores of data points and its sensor and vice-versa. Trust scores of data points are computed by taking into account data point values generated from sensor in a given neighbourhood of the given sensor. We use value similarity: the more data points referring to the same real-world event (neighbours) have similar values the higher the trust score of the data point. Trust score need to be continuously evaluated in the stream environment.

In particular, to express the trust score of sensor k at time t we use the Z-score as follows:

$$Z(X_t^{(k)}) = \frac{X_t^{(k)} - E(N_t^{(k)})}{\sqrt{\text{Var}(N_t^{(k)})}}$$

where

$$E(N_t^{(k)}) = \frac{1}{(N_t^{(k)})} \sum_{s \in N_t^{(k)}} s.value$$

and

$$\text{Var}(N_t^{(k)}) = \frac{1}{((N_t^{(k)}) - 1)} \sum_{s \in N_t^{(k)}} (s.value - E(N_t^{(k)}))^2$$

Then the can be expressed as the p-value

$$T_t^{(k)} = P(Z > Z(X_t^{(k)}))$$

Clearly, the trust values have the probability correspondence and are values in the interval $[0, 1]$.

7.5 Trust-Module architecture for IoT

In this section we present a proposal of architecture of a Trust-Module that accommodates the specifics/requirements of an IoT environment such as OpenIoT. Thus, given the fact that sensor streams in OpenIoT are by default stored in the cloud database (LSM) for further processing, implies the following architecture of the trust module:

- ✦ Trust-Module is an independent module in OpenIoT.
- ✦ It obtains the sensor streams from LSM and outputs corresponding trust streams back to LSM.

There are the following ways of computing the trust stream given available sensor streams in LSM:

1. On-line (immediate) while storing the sensor streams to LSM. The disadvantage of this approach is a heavy overloading of the capabilities of the trust module, LSM and the communication link between them.
2. Off-line on demand: computed if the corresponding query arrives. This

approach has the same disadvantage as the on-line approach, where in this case it is a blocking operation. The advantage of this approach is that LSM will not be populated with trust streams that may never be used.

3. Off-line periodically (deferred): computed periodically after the sensor streams have been stored in LSM.

We adopt the off-line solution combined with caching mechanism to optimize the computational and storage resources.

Figure 6 presents the view on the integration of the trust module in the OpenIoT architecture. Clearly, the follow of data is as follows:

- ⤴ X-GSN provides sensor streams to LSM.
- ⤴ The Trust-Module obtains the data from LSM and periodically outputs the corresponding trust streams back to LSM (stored in a separate entity that references the corresponding sensor stream).
- ⤴ Trust-Module communicates with SCH and SDUM to process queries. In particular, SCH may trigger an on-demand computation of a trust stream if this is necessary for a given query (e.g., the query specifies a minimum trust threshold for sensor data), while SDUM will monitor the performance of TM and trigger periodic computation of trust streams.

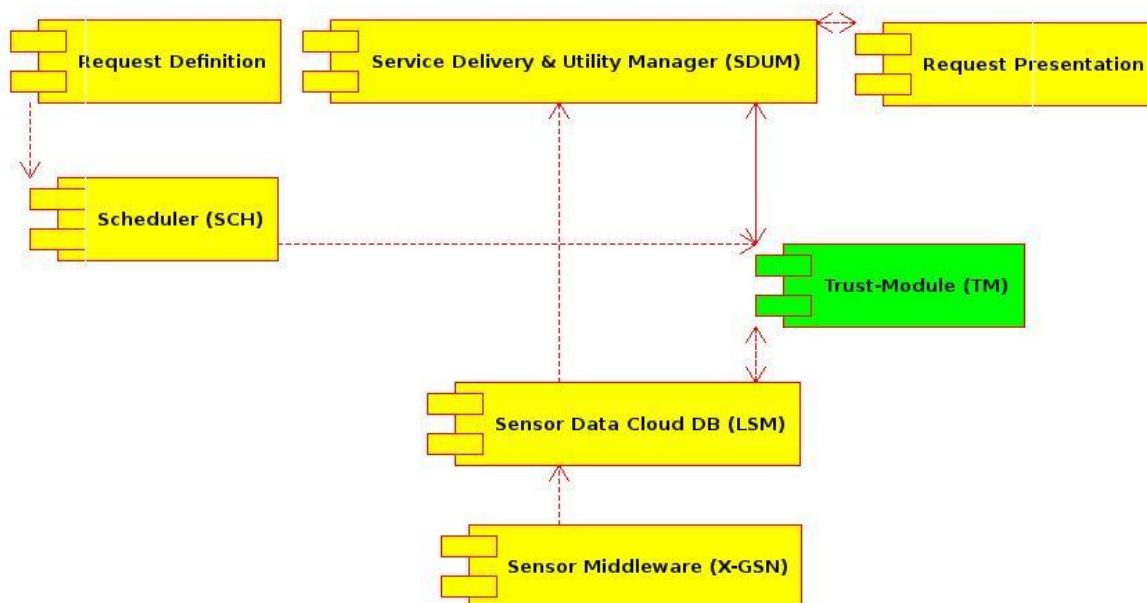


Figure 6. Trust-Module in OpenIoT.

8 IMPLEMENTATION OF THE SECURITY FRAMEWORK

In this section we give an overview of the currently implemented prototype of the security framework, including the OpenIoT CAS server, Security Client and Security management console. The documentation for how to use and configure the prototype can be found on the OpenIoT wiki².

We have adapted Jasig CAS³ for authentication and authorisation. CAS is an open source multi-protocol Single Sign-On (SSO) solution with a lot of flexibility in configuration. Particularly, it can integrate with several authentication methods such as Active Directory, JAAS, JDBC, LDAP, and so on. It can achieve high availability by providing support for storing client authentication state in distributed storage providers such as BerkleyDB, Ehcache, JDBC, Memcache, and so on. CAS can be configured to act as an OAuth2.0 server. Another OAuth provider and client library is Spring Security OAuth. However, we have opted for using CAS using OAuth wrapper because of its configuration flexibility and the ease of integration. We refer to the OpenIoT adaptation of CAS interchangeably, by *Security Server* or *OpenIoT CAS*.

8.1 General Authentication and Authorisation Approach

In this section we describe, using OpenIoT CAS, how different components of OpenIoT interact securely and how they implement access control. The OpenIoT security module consists of three modules, namely, *Security Server*, *Management Console*, and *Security Client*. In the following, we first present the overall authentication and authorisation approach and then each module is explained in details.

In the context of security module, we use the term **service** (as it is used by CAS) to refer to any component that uses the OpenIoT security module for authentication and authorisation. For example, LSM-light.server, SD&UM, Scheduler, and X-GSN are considered services. In OAuth2.0 protocol, the components that request authorisation are called clients. Therefore, the terms **service** and **client** are used interchangeably.

Each OpenIoT component and application must identify itself to the security server to be recognized as a *service* or *client*. For this purpose, the security management module provides a user interface for registering and managing services. Each service is assigned a **key** (or **clientId**) and **secret** which are considered as the credentials of the service. We distinguish between two different service types: *web application services* and *REST services*. Web application services (hereafter, simply referred to as *services*) are the applications that directly interact with users through a web interface. For example, Request Definition and Request Presentation web applications are considered as web application services. The applications that do not interact with users through a web interface and users cannot log into them from their browsers are called REST services. For example, Scheduler, SD&UM, and LSM-Server are considered as REST services.

² <https://github.com/OpenIoT/openiot/wiki/Documentation>

³ <http://www.jasig.org/cas/>

8.1.1 Authentication

In OpenIoT, user management is done centrally by the Security module. Furthermore, user authentication is the responsibility of the security server. In this way, each application does not have to individually deal with user management and authentication. In a web application, user authentication steps are the following:

1. User opens the application in her browser. For being able to access the features of the application, the user has to log in. The application, that is integrated the OpenIoT security mechanism, provides a “log in” link for the user. **Figure 7** shows the first of the security management console application. The application is accessed at <http://localhost:8080/security.management>.

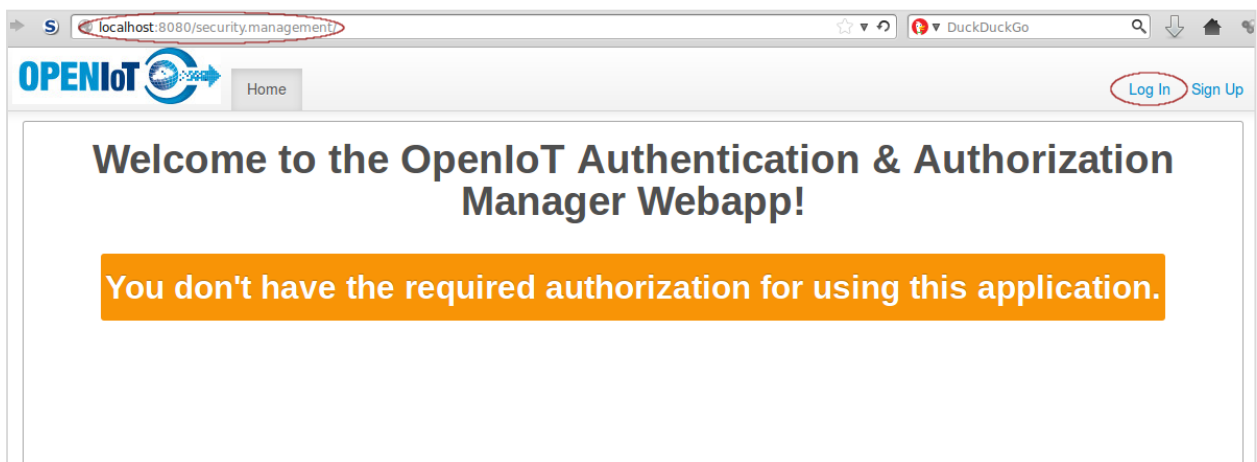


Figure 7. Home page of management console application with “Log In” link

2. The user clicked on “Log in” link. The server redirects the user’s browser to the OpenIoT CAS login page as shown in **Figure 8**.
3. The user enters her OpenIoT username and password and clicks “LOGIN”. If login is successful, the user is asked to authorize the application to access her profile, including her permission information. **Figure 9** shows the authorisation page for user “admin”.
4. If the user clicks on “Allow”, she will be redirected back to the application (**Figure 10**). In the process, the security server generates and returns a **token** to the web application. This token represents an authenticated user and is used in all the interactions between the web application and the security server or other OpenIoT components.

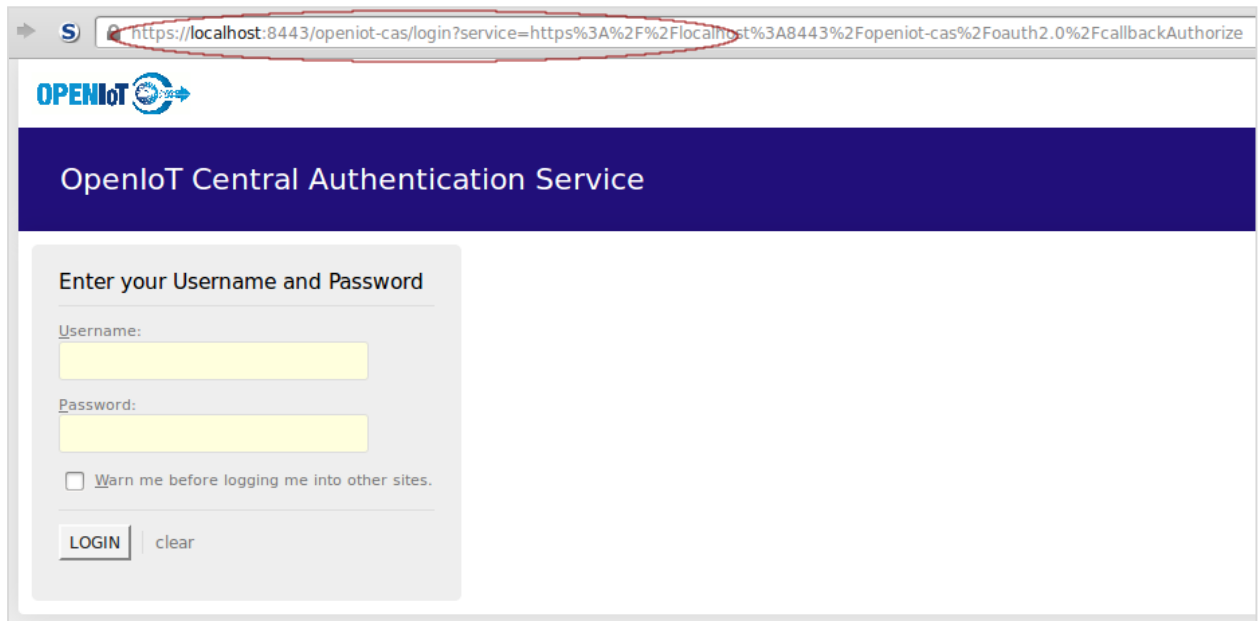


Figure 8. OpenIoT CAS login page

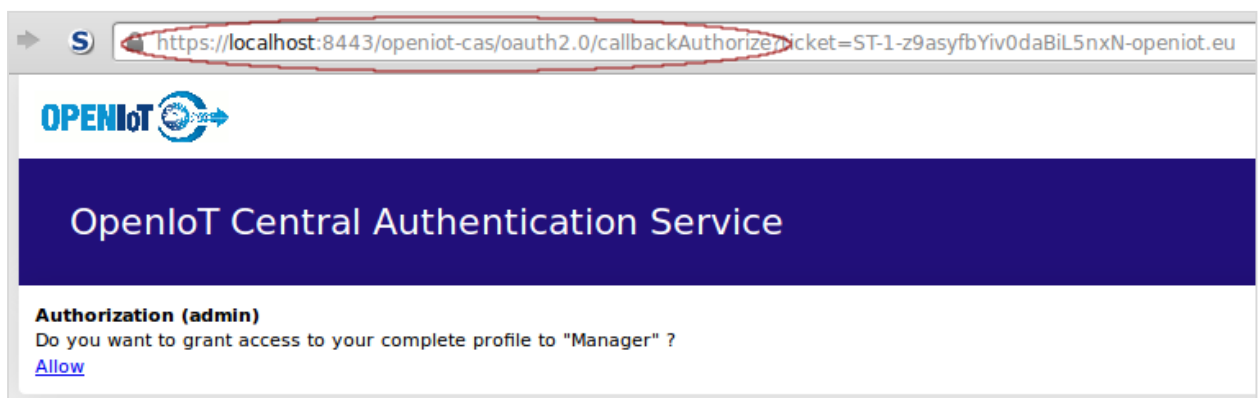


Figure 9. OpenIoT CAS authorisation page

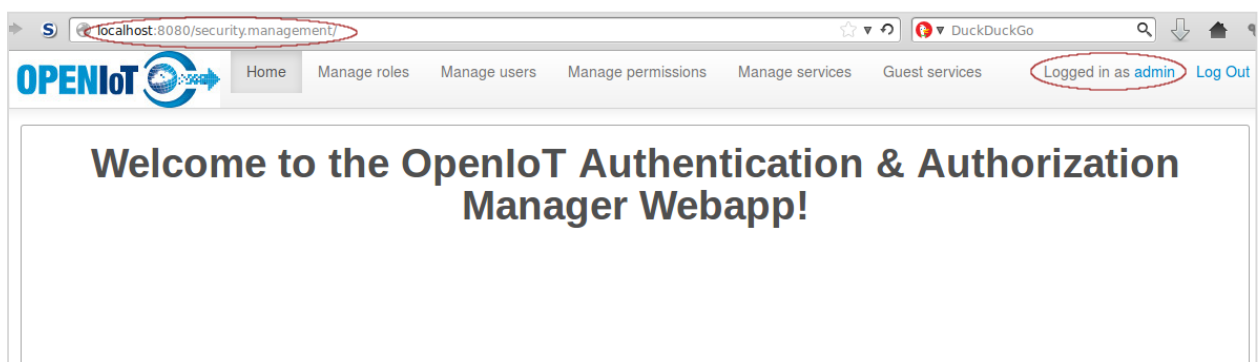


Figure 10. Home page of management console application after user is authenticated

8.1.2 Authorisation

After the user is authenticated, it is the responsibility of the application to protect

the critical resources and parts of the interface by applying an access control mechanism. The OpenIoT security module provides a unified and flexible **permission** control approach to facilitate access control. Each OpenIoT module or application defines its own set of permissions. Permissions can be selectively assigned to users. The application developer protects critical parts of code by putting permission controls before each critical part. If the user, represented by a token, does not have the required permissions, she cannot use the functionality or the resources that require those permissions.

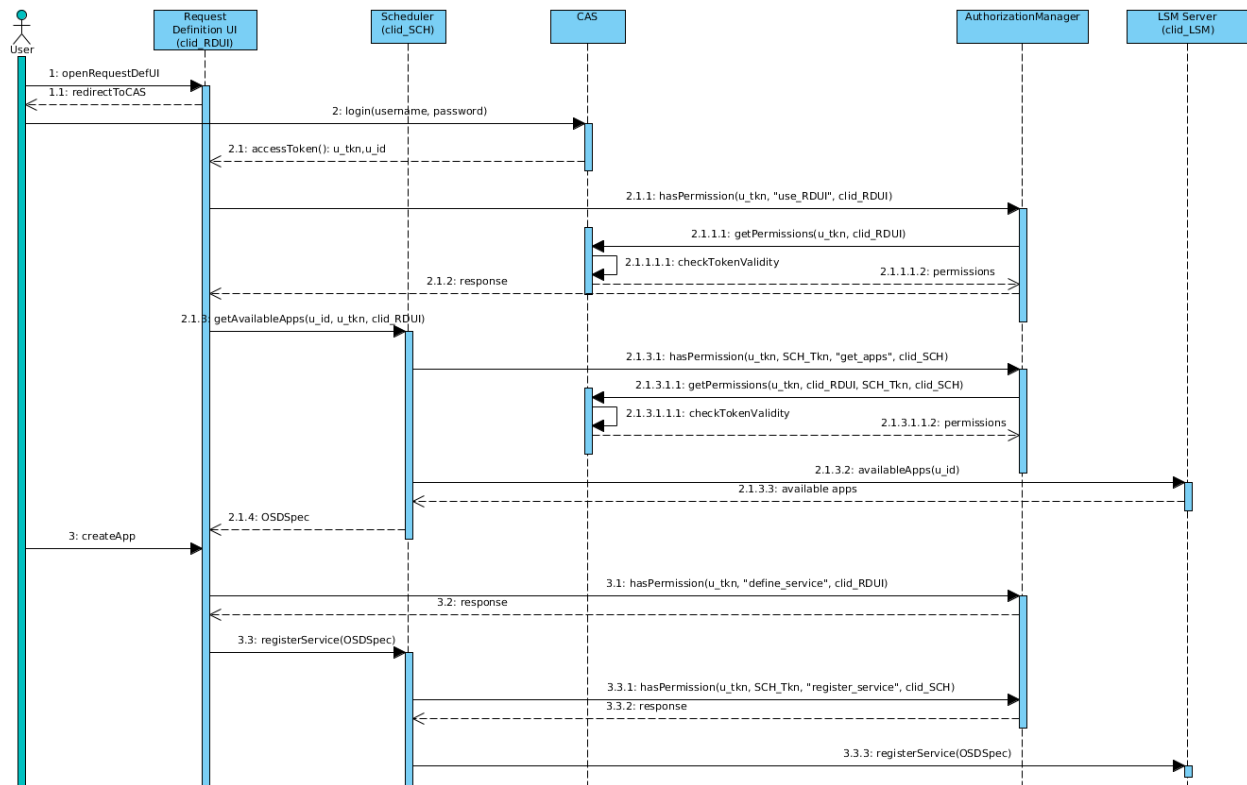


Figure 11. Sequence diagram of a sample access control

Figure 11 represents the sequence diagram of authorisation control performed by the Request Definition and Scheduler modules for two user actions. In this example, Request Definition is a web application service that is supposed to be already registered in the Security Service with **clientId** of “clid_RDUI”. Scheduler is a REST service that is supposed to be registered in the Security Server with clientId of “clid_SCH”. For improving security, in OpenIoT, we require all REST services to be authenticated in OpenIoT CAS and receive a token. In this example, we assume that Scheduler is logged in CAS and has received the token “SCH_tkn”. The first step for the user, with user ID of “u_id”, is to be authenticated as described in the previous subsection. The user token that is given to the Request Definition UI application (RDUI) is referred to as “u_tkn”. The following steps are performed to retrieve registered user applications after user authentication.

1. First, RDUI checks whether the user has the permission to use the application. This permission is represented by the permission string “use_RDUI”. For this aim, RDUI calls the *hasPermission* method of the

AuthorizationManager class, giving its *clientId*, the user token and the permission string. In this example, *AuthorizationManager* represents the Security Client utility.

2. The *AuthorizationManager* sends a request to the security server asking for permissions of the user. The request is accompanied by the user token and *clientId* of RDUI.
3. The security server verifies the validity of the token. If the token is valid (e.g., not expired), it returns the permission list of the user.
4. If the permission “use_RDUI” is among the permission list, the user is allowed to use the application. In the next step RDUI retrieves the registered applications of the user.
5. RDUI send a request to the Scheduler to retrieve the applications of the user, sending *userId*, user token, and its *clientId*.
6. Scheduler calls the *hasPermission* method of the *AuthorizationManager* class, giving its *clientId*, token, the user token, and the permission string of “get_apps”.
7. The *AuthorizationManager* sends a request to the security server asking for permissions of the user on the Scheduler service. The request is accompanied by the user token, the Scheduler token and *clientId*, and the *clientId* of RDUI (the initiator of the call).
8. The security server verifies the validity of the tokens. If the tokens are valid (e.g., not expired), it returns the permission list of the user on the Scheduler service.
9. If the permission “get_apps” is among the permissions of the user, Scheduler retrieves the applications from the LSM Server.
10. In the next step, the user wishes to register her newly created application. RDUI checks for permission “define_service”. Since the *AuthorizationManager* has already retrieved and cached the permissions of the user for RDUI, it checks the permission locally. The user application is registered only when she has the permission “define_service”.

8.2 Openlot CAS

We have adapted OAuth2.0 enabled CAS server (version 3.5.2) for authentication and authorisation in the security-server module. It has been configured to store data in the LSM-Light server.

In order to modify the configuration of the original CAS library, we use Maven2 war overlay method. This means that the files that we modify will replace the corresponding files in the standard CAS server war.

The security-server module depends on *utils.commons* and *lsm-light.client* components. Therefore, it is mandatory to install these components before trying to deploy the security-server module.

8.2.1 Configuration

The security module is deployed in JBoss and requires the SSL to be activated.

We assume that JBoss AS7 is used in standalone mode. By \$jboss-home we refer to the JBoss AS7 directory.

8.2.2 JBoss Configuration

Assume that you store the required SSL files in a path YOUR_SSL_DIR_PATH (e.g., \$jboss-home/standalone/configuration/ssl).

1. If the directory structure does not exist, run `mkdir -p YOUR_SSL_DIR_PATH`
2. Run `cd YOUR_SSL_DIR_PATH`
3. Run `keytool -genkey -alias jbosskey -keypass <password> -keyalg RSA -keystore server.keystore` Use "localhost" as Common Name [as answer to "What is your first and last name?"]. If you are deploying on a server, use the DNS name of the server instead of "localhost".
4. Run `keytool -export -alias jbosskey -keypass <password> -file server.crt -keystore server.keystore`
5. Run `keytool -import -alias jboss-cert -keypass <password> -file server.crt -keystore server.keystore` Ignore the warning!
6. In \$jboss-home/standalone/configuration/standalone.xml add the following connector in `<subsystem xmlns="urn:jboss:domain:web:1.1" ..`

```
<connector name="https" protocol="HTTP/1.1" scheme="https" socket-binding="https" secure="true">
<ssl name="https" key-alias="jbosskey" password=<password> certificate-key-file="YOUR_SSL_DIR_PATH/server.keystore" />
</connector>
```

7. Next, you'll have to import this certificate into the java trust-store with the command `keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts -file server.crt -alias incommon`. In Linux you will have to do this as root.
8. Restart JBoss and go to `https://localhost:8443` to see if SSL is enabled and works correctly.

8.2.2.1 OpenIoT CAS Configuration

Configuration parameters for the security server module can be found in the OpenIoT global properties file (openiot.properties). The relevant properties are the following:

- *security.lsm.sparql.endpoint*: the LSM SPARQL endpoint
- *security.lsm.graphURL*: the graph in which data will be stored
- *server.name*: the server address on which the OpenIoT CAS is deployed

- The lifetime of tickets can be configured by two properties: 1) ***tgt.maxTimeToLiveInSeconds*** that specifies the maximum time to live of the ticket. After this amount of time, the ticket will expire no matter how many times it has been accessed; 2) ***tgt.timeToKillInSeconds*** that determines the the maximum time during which the ticket can remain idle. If no accesses occur after this period, the ticket will expire.
- *Parameters for exporting initial data into LSM:* when the OpenIoT CAS server is deployed for the first time, if its data graph is empty, by default some bootstrapping data will be stored in the graph. This data includes the administrator user information and its default roles and permissions, as well as the default services. The relevant properties are the following:
 - *security.automaticServiceSetup*: indicates whether or not the bootstrapping should be done if the data graph is empty. The default value is **true**.
 - *security.initialize.admin.username*: username of the admin user. The default value is **admin**.
 - *security.initialize.admin.password*: password of the admin user. The default value is **secret**.
 - *security.initialize.admin.email*: email of the admin user. The default value is **admin@openiot.eu**.
 - *security.initialize.lsmserver.username*: default username for LSM-light.server. The same property exists for *scheduler* and *sdum* modules.
 - *security.initialize.lsmserver.password*: default password for LSM-light.server. The same property exists for *scheduler* and *sdum* modules.
 - *security.initialize.cas.prefix*: the prefix for the CAS service. If you are not deploying on localhost, change the address (or port).
 - *security.initialize.management.prefix*: the prefix for the security management console service. If you are not deploying on localhost, change the address (or port).
 - *security.initialize.management.key*: the key for the security management console service.
 - *security.initialize.management.secret*: the secret for the security management console service.
 - *security.demoWebappServices*: the list of web application services that are created for each demo user (if enabled by setting *security.automaticServiceSetup* to true) upon signing up.
 - *security.demoServices*: the list of all demo services that are enabled for demo users if this feature is activated as mentioned above.

The following services are created in the bootstrapping step:

- **ServiceManager:** This service is required for service management in CAS. The hostname and port in the *serviceld* column can be changed if

required.

- **HTTP:** This service is required for OAuth2.0 support in CAS. The hostname and port in the `serviceld` column can be changed if required.
- **lsm-server:** This service is defined for the LSM Server.
- **openiot-security-manager-app:** This service corresponds to the management console of the security&privacy module. We will explain how to manage these services later in this documentation.

8.2.3 Deployment

For deploying the OpenIoT CAS server war in JBoss, go to `security-server` module directory and run `mvn jboss-as:deploy`. If the deployment is successful, you must be able to see a web application called `openiot-cas` deployed in JBoss. Go to <https://localhost:8443/openiot-cas> and log in using the default admin credentials specified in `openiot.properties` file to verify the deployment.

8.2.4 Managing Services in OpenIoT CAS

Each application has to be registered in OpenIoT CAS as a service in order to be recognized for authentication and authorisation. The OpenIoT security management console provides a web interface for registering new services and managing the existing ones.

8.3 Security Management Console

The OpenIoT Security and Privacy module comes with a management console for managing services, users, and permissions.

8.3.1 Configuration

By default, it is assumed that the management console will be deployed in a local JBoss instance and is accessible on <http://localhost:8080/security.management/>. If this is not the case in your setting, you must adapt the `clients.callbackUrl` property in `web-client.ini` file, which is located in `security-management/src/main/resources` directory. Note that the recommended way to modify configuration parameters is to copy the `web-client.ini` to `$jboss.config.dir/web-client-security.management.ini` and modify the desired values there.

Likewise, it is assumed that the OpenIoT CAS is deployed locally and is accessible on <https://localhost:8443/openiot-cas>. If this does not correspond to your setting, update `casOAuthClient.casOAuthUrl` and `clientsRealm.permissionsURL` properties in the ini configuration file.

Finally, it is recommended that you change the default secret that is used for `openiot-security-manager-app` (in the ini configuration file and/or in the global `openiot.properties` file).

The location of log file and other logging configuration, such as logging levels, can be modified in `security-management/src/main/resources/logback.xml`. By default, the

configured log file is `$jboss-home/standalone/log/security-management.log`.

8.3.2 Deployment

For using the management console, security-server module must be deployed. Instructions for deploying the security-server module are provided in Section 8.2 . This web application is registered in the OpenIoT CAS by the name of “openiot-security-manager-app”.

You can deploy the security console web application by running `mvn jboss-as:deploy` in security-management module directory. After successful deployment, go to <http://localhost:8080/security.management/> for using security module management console. Log in with username “admin”, and password “secret” (the default credentials of the admin user). This user has all the necessary permissions to manage users and permissions for all registered services. **Figure 10** shows the first page of the management console application after successful login.

8.3.3 Use

The management console provides the following functionality:

- Defining/managing permissions for registered services
- Defining/managing roles for registered services
- Assigning roles to users or revoking roles from users
- Adding/managing services
- Registering/managing users

Role is a container for one or more permissions. Permissions cannot be directly assigned to users. Instead, they must be assigned to a role. Then the role is granted to the users. Roles and permissions are defined per service. This means that for each service, we have to define its own set of permissions and roles. We use Apache Shiro's⁴ wildcard permission syntax to define the permissions. We briefly describe the syntax in the following. Please refer to the official documentation⁵ for detailed information about wildcard permissions.

A wildcard permission, in the simplest case, is a string permission like “register_service” or “use_app”. In this case, the developer checks whether or not the user has the exact permission specified by the permission string. The wildcard permission “*”, which means *all permissions*, can be assigned to users. This special permission string represents all possible permission strings. Wildcard permissions can contain multiple *levels*. Each level is separated by a “:”. For example, the permission string “use_app:part_n” has two levels: “use_app” and “part_n”. The number of levels is unlimited and the semantic of levels is defined by the permission designer. The wildcard character can be used in any level. For example a user with permission “use_app:*” has any permission like “use_app:X”, where “X” can be any permission string. Last but not least, each level can contain

⁴ <http://shiro.apache.org>

⁵ <http://shiro.apache.org/permissions.html>

multiple permissions. For example, assigning the permission “use_app:part1, part2” to a user is equivalent to assigning her “use_app:part1” and “use_app:part2” permissions.

8.3.3.1 Managing Services

In the “Manage services” section of the management console, users can view registered services, edit, or remove them. New web application services and rest services can also be defined in this section. **Figure 12** shows an screenshot of the “Manage services” section.

Available Services				
key	Secret	URL	Display name	Options
scheduler	scheduler.secret	REST://scheduler	scheduler_	
lsm-server	lsm-server.secret	http://lsm-light.server/callback	LSM-Server	
openiot-security-manager-app	openiot-security-manager-app-secret	http://services.openiot.eu:8080/sec	Manager	
lsm-client	lsm-client.secret	REST://lsm-light.client	LSM-Client	
sdum	sdum.secret	REST://sdum	sdum_	
requestDefinitionUI	requestDefinitionUI-secret	http://services.openiot.eu:8080/ui.r	RequestDefinition	
requestPresentationUI	requestPresentationUI-secret	http://services.openiot.eu:8080/ui.r	RequestPresentation	

Figure 12. Managing services in management console application

To add a new web application service, click on the “New Service” button and specify the following fields as shown in Figure 13.

- In the **Key** field enter the clientId of the service.
- In the **Secret** field enter the secret of the service.
- In the **URL** field enter the callback URL of the service. For example, in case of management console service deployed on localhost, we enter http://localhost:8080/security.management/callback?client_name=CasOAuthWrapperClient. This URL consists of the application path and the mandatory /callback?client_name=CasOAuthWrapperClient part.
- In the **Display name** field enter the name of the service that is displayed to the user in the authorisation confirmation page while logging in any web application that uses the OpenIoT CAS for authentication and authorisation. For example, in case of the management console, the default display name is "Manager".

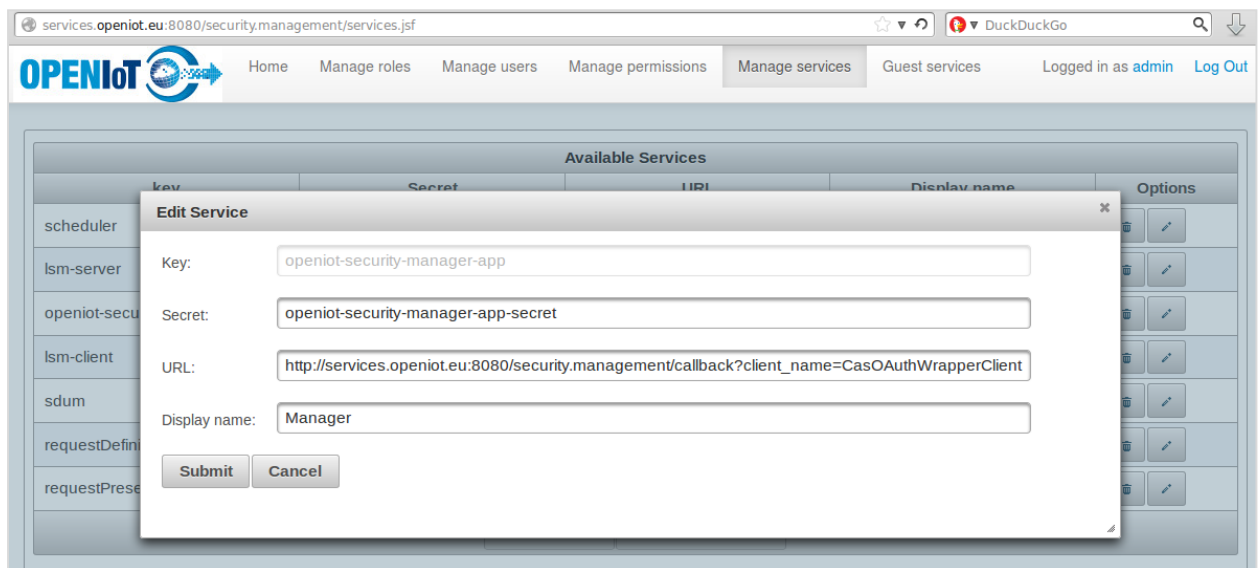


Figure 13. Adding/editing web application services

To add a new REST service, click on the “New REST Service” button and specify the following fields as shown in Figure 14.

- In the **Key** field enter the clientId of the service.
- In the **Secret** field enter the secret of the service.

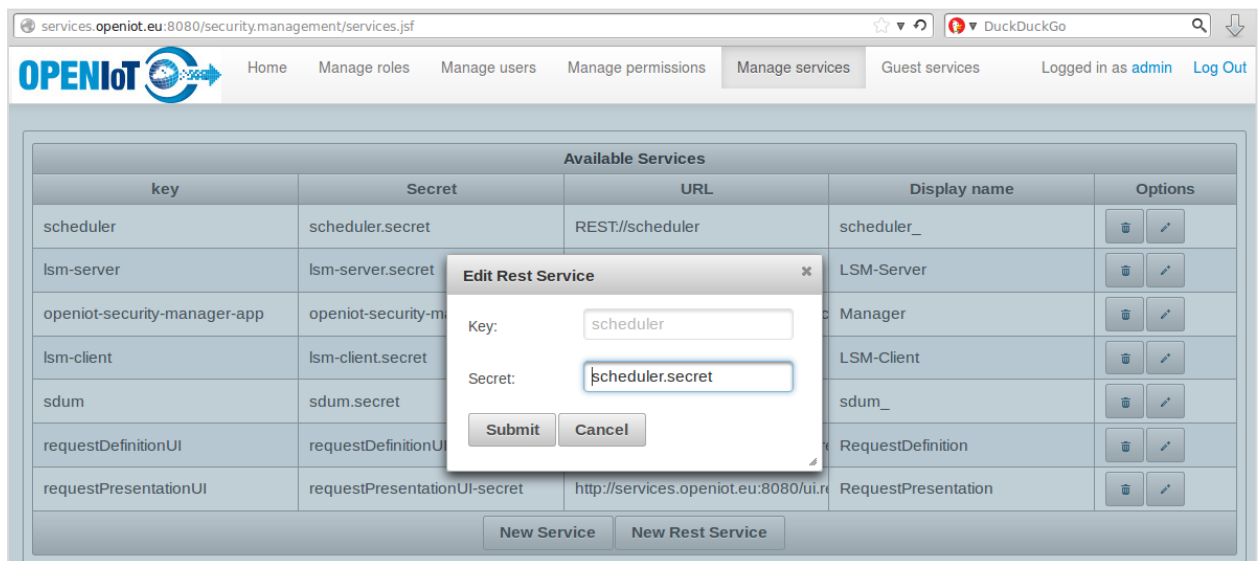


Figure 14. Adding/editing REST services

8.3.3.2 Managing Permissions

In the “Manage permissions” section of the management console application, users can manage permissions for registered services. In the first step, the user must select a service to manage its permissions as shown in **Figure 15**.

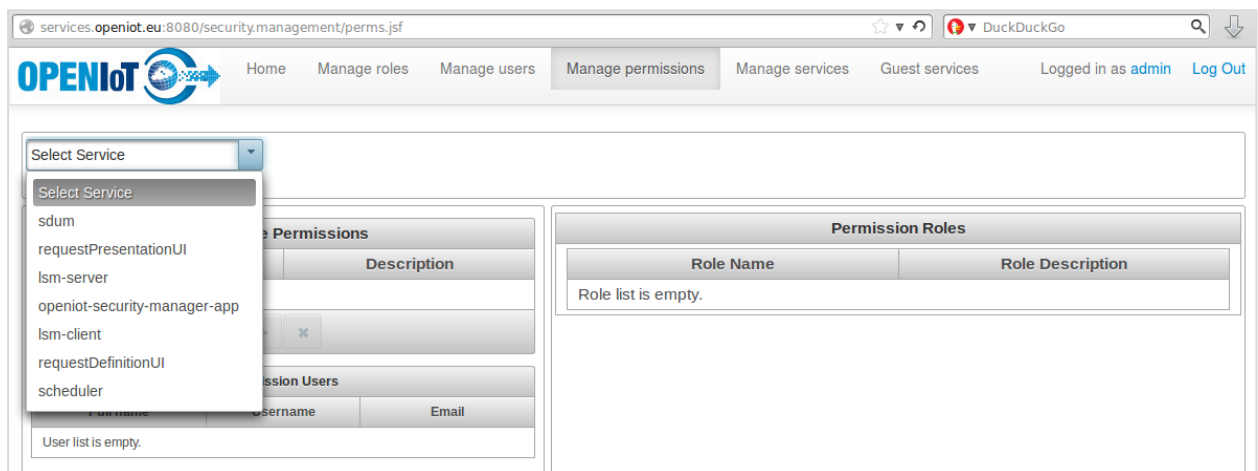


Figure 15. Permission management section of management console

After selecting the service, the list of permissions defined for the service is shown. By selecting permission, the list of roles that contain the selected permission is populated. In addition, the list of users who have the selected permission is updated. The selected permission can be deleted given that the user has the permission “admin:delete_permission:openiot-security-manager-app”. Figure 16 shows a snapshot of the application, when the “lsm-server” service is selected for managing its permissions.

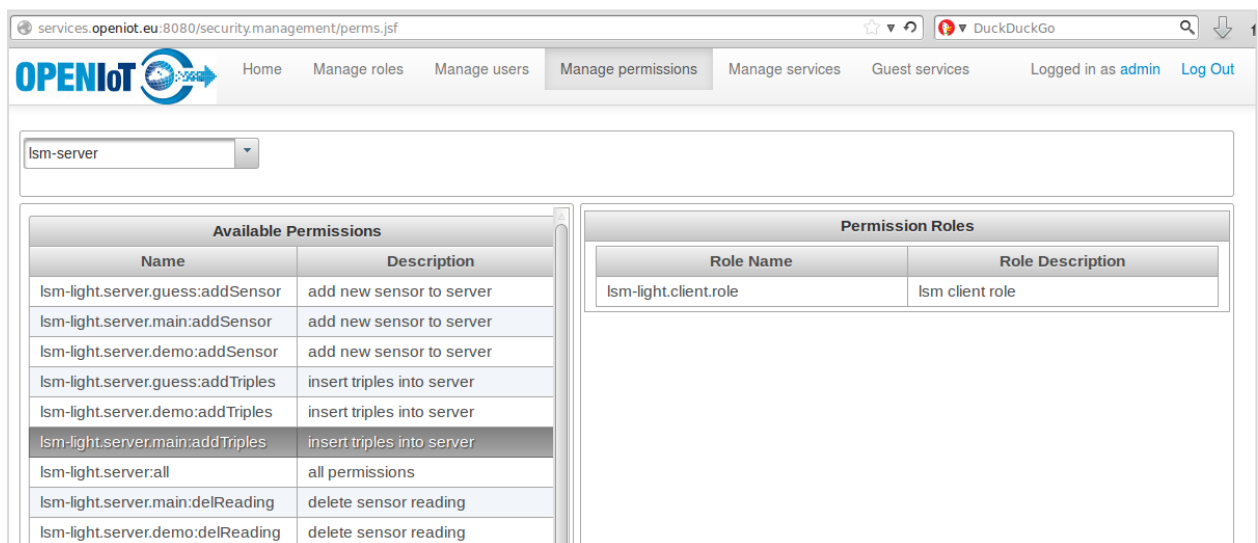


Figure 16. The list of permissions defined for the "lsm-server" service

Users can define new permissions for the selected service, provided that they have the permission “admin:create_permission:openiot-security-manager-app”. As shown in Figure 17, a new permission is defined by its name. The description of the permission helps to document the purpose of the permission.

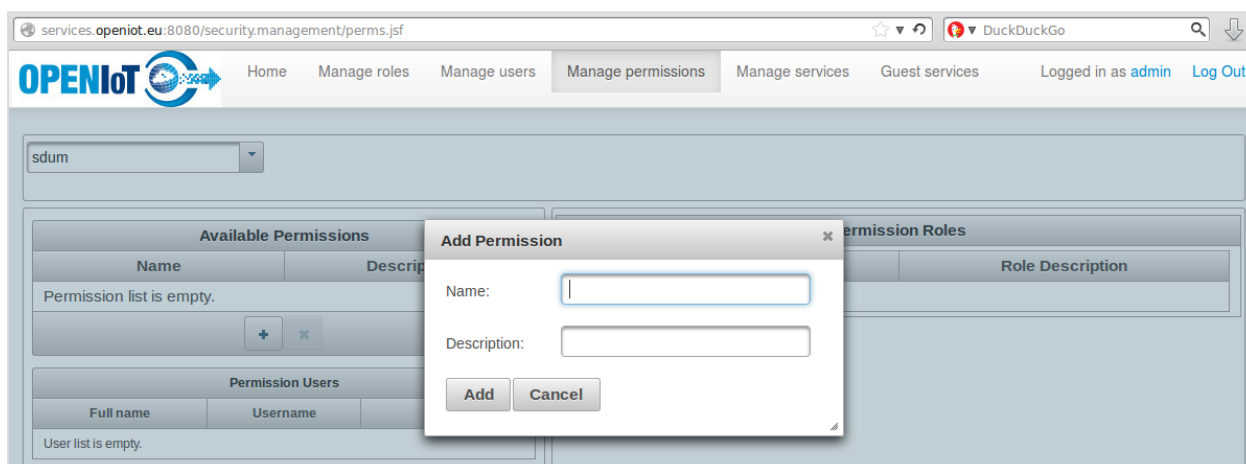


Figure 17. Adding new permission for the selected service

8.3.3.3 Managing Roles

In the “Manage roles” section of the management console application, users can manage roles for registered services. In the first step, the user must select a service to manage its roles. As shown in **Figure 18**, after selecting a service, its defined roles are shown. By selection a role, the permissions assigned to it are displayed in the right pane. The list of users who are granted the selected role is also shown. In this page, new permissions can be assigned to the selected role; the role can be granted to more users; the role can be revoked from users; and permissions can be revoked from the role. The selected role can be deleted given that the user has the permission “admin:delete_role:openiot-security-manager-app”.

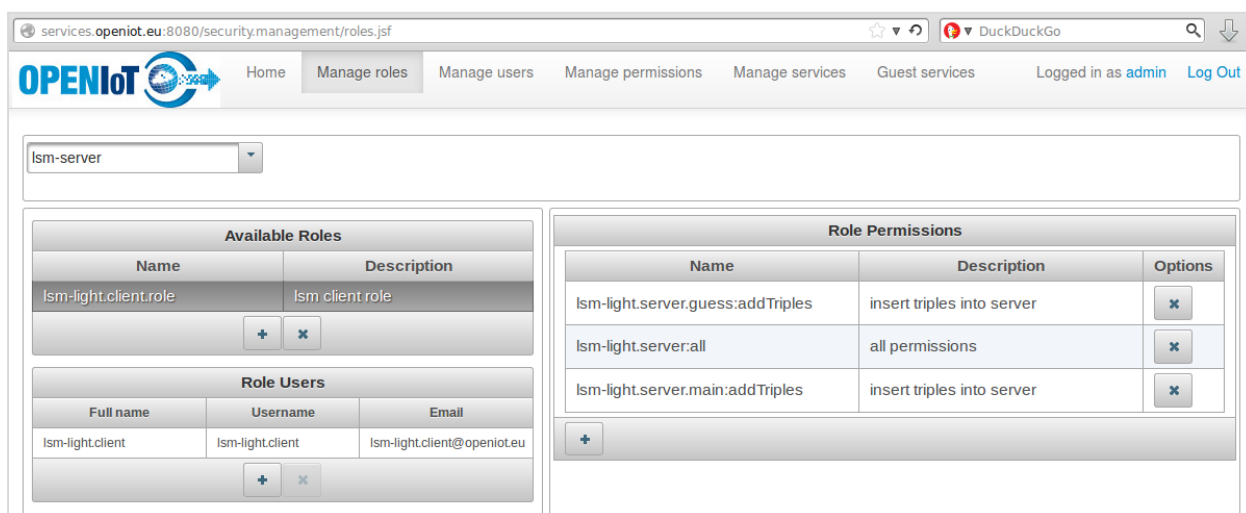


Figure 18. Role management in management console application

Users can define new roles for the selected service, provided that they have the permission “admin:create_role:openiot-security-manager-app”. A new role is defined by its name. The description of the permission helps to document the purpose of the permission.

8.3.3.4 User Management

In the “Manage users” section of the management console application, users can

see the registered users and their granted roles for registered services. As shown in **Figure 19**, after selecting a user and a service, her granted roles are shown. By selection a role, the permissions assigned to it are displayed in the right pane. New roles can be granted to the selected user and roles can be revoked from the selected user. The selected user can be deleted given that the logged in user has the permission “admin:delete_user”.

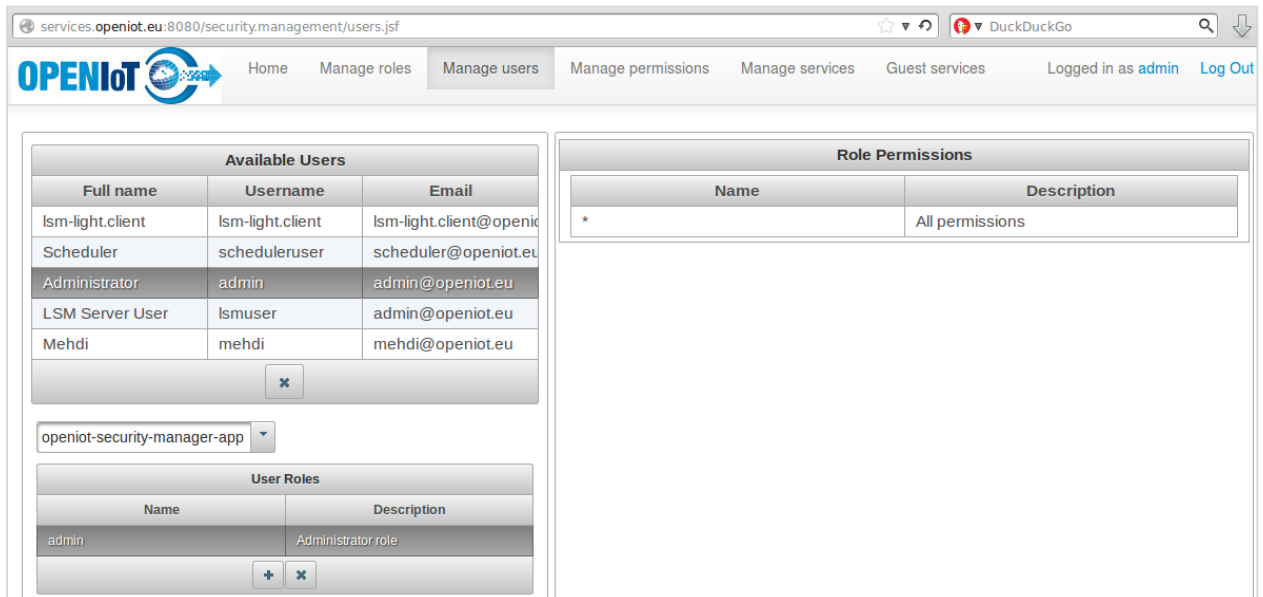


Figure 19. User management in management console application

8.3.3.5 User Registration

A new user can sign up using the management console application (**Figure 20**). If the `security.automaticServiceSetup` property is set to true in the `openiot.properties` file, then the user can enter a Service URL Prefix so that the required services are generated for her standard OpenIoT web application modules which are deployed locally. The list of these modules must be defined in the `openiot.properties` file using the property `security.demoWebappServices`. The format is like **[webapp1 name, key1, secret1, display name][webapp2 name, key2, secret2, display name]**, etc. If the property `security.signup.useCaptcha` is set to true, a captcha field is added to the signup form.

8.3.3.6 Demo Services

If the property `security.automaticServiceSetup` is set to true in the `openiot.properties` file, users can see the list of the services that are configured for demo usage in the “Guest services” section. The property `security.demoServices` indicates the services that are accessible as demo services separated by a comma. For REST services the service key should be put in this comma separated list. For web application service the name of the webapp must be placed in the list.

Figure 20. User registration page

8.3.3.7 Management Console Permissions

For using the security management console application, users need to have the permission “admin:user_mgmt_general”. This means that without this permission the user doesn't have access to the functionality of the security management application. The list of other service-specific permissions is provided in **Table 5**.

Name	Description
<i>admin:delete_role:SERVICE_NAME</i>	Permission for deleting roles for service 'SERVICE_NAME'
<i>admin:create_role:SERVICE_NAME</i>	Permission for creating new roles for service 'SERVICE_NAME'
<i>admin:grant_role:SERVICE_NAME</i>	Permission for granting/revoking roles to users for service 'SERVICE_NAME'
<i>admin:create_permission:SERVICE_NAME</i>	Permission for creating new permissions for service 'SERVICE_NAME'
<i>admin:delete_permission:SERVICE_NAME</i>	Permission for deleting permissions for service 'SERVICE_NAME'
<i>admin:user_mgmt:SERVICE_NAME</i>	User management permission for service 'SERVICE_NAME'

Table 5. Permissions for working with the management console application

For example, if you want to grant all permissions to the user *openiotuser* to manage roles and permissions for the service *openiot-security-manager-app*, you can create a role named “user-manager” in service *openiot-security-manager-app* and add the above permissions to this role. Then assign this role to *openiotuser*. Note that in this case, in the above permissions, **SERVICE_NAME** must be replaced by *openiot-security-manager-app*.

8.4 Security Client

The Security Client module provides utilities to facilitate authentication and authorisation in other OpenIoT components and applications. The OpenIoT security client can be used in web applications that provide a user interface and interact directly with users. It can also be used in intermediate service providers that interact with other components through WebServices or REST calls. In order to be able to use this module for authentication and access control, some specific configurations must be done in the modules. The OpenIoT security management console uses the security client module for authentication and authorisation. It is a good example of how to use this module in new OpenIoT applications.

8.4.1 Deployment

In order to use the security client in the modules that are built using maven, it must be built and installed in the local maven repository. For building and installing the security module, go to the security-client module directory and run `mvn install`. The security client depends on the `utils.commons` module. Therefore, it must be installed in the local maven repository before building and installing the security module.

8.4.2 Maven Dependencies

The first step in using the security client is to add the following dependency to the pom file of the module:

```
<dependency>
  <groupId>org.openiot</groupId>
  <artifactId>security.client</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The default client library that we use for authentication & authorisation using OpenIoT CAS is Apache Shiro. This library is provided as a dependency in the security module. Therefore, it is not necessary to explicitly add it as a dependency in the pom files.

8.4.3 Usage for Web Applications

The security client can be used in web applications that provide *log in* for users and implement access control.

8.4.3.1 Shiro Webapp Configuration

In order to initialize Shiro in a web application, add the following XML chunk to your *web.xml*:

```

<context-param>
    <param-name>shiroEnvironmentClass</param-name>
    <param-
value>org.openiot.security.client.CustomIniWebEnvironment</param-value>
</context-param>
<context-param>
    <param-name>module-name</param-name>
    <param-value>MODULE_NAME</param-value>
</context-param>
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
<listener>
    <listener-
class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>
</listener>

```

Note that in *web.xml* all other configuration places, `MODULE_NAME` should be replaced by the actual name you give to your module. For configuring Shiro, we can create a file named *web-client.ini* with the following example contents (taken from the security management console module) and place it into the classpath (e.g. in `WEB-INF` directory). However, the recommended approach is to create a configuration file named *web-client-MODULE_NAME.ini* in the JBoss configuration folder. In fact, if this file is not found, the security client will try to find a *web-client.ini* file in the classpath of your module.

```

[main]
casOAuthClient=org.pac4j.oauth.client.CasOAuthWrapperClient
casOAuthClient.casOAuthUrl=https://localhost:8443/openiot-cas/oauth2.0
casOAuthClient.key=openiot-security-manager-app
casOAuthClient.secret=openiot-security-manager-app-secret
# Sets the callbackUrl for each client in the list
clients = org.pac4j.core.client.Clients
clients.callbackUrl = http://localhost:8080/security.management/callback
clients.clientsList = $casOAuthClient
clientsFilter = org.openiot.security.client.CasOAuthClientFilter
clientsFilter.clients = $clients
clientsFilter.failureUrl = /error.xhtml
casOAuthRoles = io.buji.pac4j.filter.ClientRolesAuthorizationFilter
casOAuthRoles.client = $casOAuthClient
casOAuthUsers = io.buji.pac4j.filter.ClientUserFilter
casOAuthUsers.client = $casOAuthClient
clientsRealm = org.openiot.security.client.CasOAuthClientRealm
clientsRealm.permissionsURL=https://localhost:8443/openiot-
cas/oauth2.0/permissions
clientsRealm.defaultRoles = ROLE_USER
clientsRealm.clients = $clients
sessionManager = org.apache.shiro.web.session.mgt.DefaultWebSessionManager
sessionManager.globalSessionTimeout=1800000
securityManager.sessionManager = $sessionManager

[urls]
# NOTE: Order matters! The first match wins.
/logout = logout
/callback = clientsFilter
/signup.xhtml = anon
/error.xhtml = anon
/index.* = anon
/index = anon
/home.* = anon
/home = anon
/javax.faces.*/** = anon
/*?/* = casOAuthRoles[ROLE_USER]

```

The most important properties in this configuration file are the following:

- ***casOAuthClient.casOAuthUrl*** property specifies the address of the OAuth provider to which users will be redirected for authentication. This

address refers to the OpenIoT CAS server. If the OpenIoT CAS is deployed in a different place, this URL must be updated accordingly.

- ***casOAuthClient.key*** designates the clientID of the web client, and ***casOAuthClient.secret*** specifies its secret. If you specify the properties *casOAuthClient.MODULE_NAME.key* and *casOAuthClient.MODULE_NAME.secret* in the global *openiot.properties* file, you can skip providing *casOAuthClient.key* and *casOAuthClient.secret* in the ini file; because they will be overridden by the values from the *openiot.properties* file.
- ***clients.callbackUrl*** is the URL that is called back by the CAS server. It consists of the URL of your application followed by */callback*.
- ***clientsRealm.permissionsURL*** is the URL on the OAuth provider for retrieving permissions of the authenticated user. If the OpenIoT CAS is deployed in a different place, this URL must be updated accordingly.
- ***sessionManager*** is responsible for managing user sessions and must be set according to the above example ini configuration file.

In the **[urls]** section, we can define protection rules using URL patterns. In the sample configuration file, */signup.xhtml*, */index.**, */javax.faces.*/***, and */home.** are configured to be accessible to users without requiring them to log in. */?*/** = casOAuthRoles[ROLE_USER]* signifies that all other resources are protected and are accessible only to authenticated users. If the user has not been logged in yet, she will be redirected to the OpenIoT CAS login page. */logout = logout* and */callback = clientsFilter* must be left as they are to provide logout capability and OAuth authentication. For more information on configuring Shiro please refer to its official documents⁶.

8.4.3.2 Manual Redirection

Using access control rules in the **[urls]** section of the Shiro configuration file is the easiest way to handle access to different resources of the web application. However, this can also be performed manually, for example in a Servlet. The following code snippet in a servlet checks whether or not the user has already been authenticated. If not, it redirects the user to the login page (in our case, it redirects the user to the login page on the OpenIoT CAS).

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.subject.Subject;
import org.openiot.security.client.AccessControlUtil;
...
Subject subject = SecurityUtils.getSubject();
if (!subject.isAuthenticated())
    AccessControlUtil.getInstance().redirectToLogin(req, resp);
```

⁶ <http://shiro.apache.org/web.html>

8.4.3.3 Checking for Permissions and Roles

Having made sure that the user has been authenticated, it is straightforward to check if the user has a role or permission for the given service (e.g., current web application) using the `org.openiot.security.client.AccessControlUtil` utility class:

1. `hasPermission(String perm)`: checks whether or not the user has the given permission
2. `hasRole(String role)`: checks whether or not the user has the given role

In order to use the utility methods provided by `AccessControlUtil`, first we need to obtain a reference to its (singleton) object by calling the `AccessControlUtil.getInstance()` method. Then you can call the desired methods on this reference as shown in the following example:

```
...
for (RegisteredService registeredService : services) {
    String name = registeredService.getName();
    // Checking access
    if (AccessControlUtil.getInstance().hasPermission("admin:user_mgmt:" +
name))
        allServices.put(registeredService.getId(), registeredService);
}
...
```

`AccessControlUtil.getInstance().getOAuthAuthorizationCredentials()` returns an instance of `OAuthAuthorizationCredentials` that contains the *userId* and *accessToken* of the authenticated user, and the *clientId* of the web application.

8.4.3.4 Tag libraries

A set of JSP and JSF tag libraries are provided to facilitate using authentication and authorisation in OpenIoT applications. We explain how to use JSF taglib here. Using JSP taglib is similar.

In order to be able to use JSF taglib in an XHTML page, we need to declare the *openiot* namespace by adding `xmlns:openiot="http://openiot.org/tags"` to the *html* tag:

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:p="http://primefaces.org/ui"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:openiot="http://openiot.org/tags">
```


The available tags are presented in **Table 6**:

<i>Tag</i>	<i>Description</i>
<code><openiot:hasPermission name="PERM_NAME"></code>	Renders the tag body if the user has the permission specified by PERM_NAME
<code><openiot:lacksPermission name="PERM_NAME"></code>	Does not render the tag body if the user lacks the permission specified by PERM_NAME
<code><openiot:hasRole name="ROLE_NAME"></code>	Renders the tag body if the user has the role specified by ROLE_NAME
<code><openiot:laksRole name="ROLE_NAME"></code>	Does not render the tag body if the user lacks the role specified by ROLE_NAME
<code><openiot:hasAnyRoles name="ROLE_NAME1, ROLE_NAME2, ..."></code>	Renders the tag body if the user has at least one of the roles specified by ROLE_NAME1, ROLE_NAME2, etc.

Table 6. JSF tags provided by the the security client module

The security client module also provides JSF tags for Apache Shiro (shiro-faces). In order to use these tags in your XHTML document, declare Shiro's namespace in your documents by adding `xmlns:shiro="http://shiro.apache.org/tags"` to the *html* tag. The following tags can be used in your JSF pages (other Shiro tags are not recommended to be used):

- **<shiro:user>** renders the tag body if the user is authenticated
- **<shiro:guest>** renders the tag body if the user is not authenticated
- **<shiro:principal type="org.pac4j.oauth.profile.casoauthwrapper.CasOAuthWrapperProfile" property="id"/>** displays the username of the user if the user is authenticated

8.4.4 RESTful Usage

The security client module can be used in modules that do not directly interact with users but need to implement access control to fulfil user requests.

8.4.4.1 Configuration

In order to use the security client, first obtain an instance of `org.openiot.security.client.AccessControlUtil` configured for RESTful authentication and authorisation. This can be done by calling one of the following methods:

1. `AccessControlUtil.getRestInstance("module-name", "config-dir")`
2. `AccessControlUtil.getRestInstance("module-name")`
3. `AccessControlUtil.getRestInstance()`

If one of the methods that take "module-name" as parameter is used, then the security module first looks for a file named *rest-client-MODULE_NAME.ini* in the configuration directory. "module-name" is the name of the module that is using the security client. MODULE_NAME refers to the same name. If the configuration directory is not specified by passing the "config-dir" parameter to the method, the JBoss configuration folder is considered as the configuration directory. If the method with no parameter is called for obtaining an instance of the *AccessControlUtil* class, then you must create a file named *rest-client.ini* in the classpath. If *rest-client-MODULE_NAME.ini* is not found, in case of using one the first two methods, the security client falls back to *rest-client.ini* in the classpath for reading the configuration parameters. The content of the configuration file should be similar to the following example. The values must be modified to conform to the setting of your module.

In this configuration file, the following parameters are to be modified by the module developers if needed:

- ***casOAuthClient.casOAuthRestUrl*** property specifies the address of the OAuth provider for RESTful authentication. It refers to the address of the OpenIoT CAS server. If the OpenIoT CAS is deployed in a different place, this URL must be updated accordingly.
- ***casOAuthClient.key*** designates the *clientId* of the service and ***casOAuthClient.secret*** specifies its secret.
- ***clientsRealm.permissionsURL*** is the URL on the OAuth provider for retrieving permissions of the authenticated user. If the OpenIoT CAS is deployed in a different place, this URL must be updated accordingly.

```
[main]
casOAuthClient = org.openiot.security.client.rest.CasOAuthWrapperClientRest
casOAuthClient.casOAuthUrl = https://localhost:8443/openiot-cas/oauth2.0
casOAuthClient.casOAuthRestUrl = https://localhost:8443/openiot-cas/openiot1/tickets
casOAuthClient.key = testservice1
casOAuthClient.secret = testsecret1
clients = org.pac4j.core.client.Clients
clients.callbackUrl = http://localhost:8080/sth/callback
clients.clientsList = $casOAuthClient
clientsRealm = org.openiot.security.client.rest.CasOAuthClientRealmRest
clientsRealm.permissionsURL = https://localhost:8443/openiot-cas/oauth2.0/permissions
clientsRealm.defaultRoles = ROLE_USER
clientsRealm.clients = $clients
```

If the method that takes the name of your module as parameter is used to obtain an instance of the *AccessControlUtil* class, then you can provide the key and secret of your client in the global *openiot.properties* file as follows:

- `casOAuthClient.key.MODULE_NAME=CLIENT_KEY`
- `casOAuthClient.secret.MODULE_NAME=CLIENT_SECRET`

`CLIENT_KEY` and `CLIENT_SECRET` should be replaced by the key and secret of your module. If these properties are found in *openiot.properties*, they override the key and secret specified in the ini configuration file.

8.4.4.2 SSL Troubleshooting

If the security server (the OpenIoT CAS) is deployed on a host other than the localhost and the server has a self-signed certificate, you might get the following error:

```
javax.net.ssl.SSLPeerUnverifiedException:      peer      not
authenticated
```

In this case, you need to add the server's certificate to the list of trusted certificates of the JVM. One way of doing that is by performing the following steps:

1. `echo -n | openssl s_client -connect SERVER_ADDRESS:8443 |
sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' >
SERVER_ADDRESS.cert`
2. `keytool -importcert -alias "SERVER_ADDRESS" -file
SERVER_ADDRESS.cert -keystore
JRE_PATH/lib/security/jssecacerts`

Replace *SERVER_ADDRESS* with the address of the server and *JRE_PATH* with the path of your *JRE*. If *jssecacerts* does not exist, use *cacerts* instead.

8.4.4.3 Authentication and Authorisation

Having obtained, for example in a variable named `restInstance`, an instance of `org.openiot.security.client.AccessControlUtil` configured for RESTful authentication and authorisation, the following functionality is accessible.

- For logging in and obtaining a token, use `restInstance.login()` method and provide your username and password.
- Use `restInstance.getOAuthAuthorizationCredentials()` to obtain an `OAuthAuthorizationCredentials` object containing the *acesstoken*, *clientId*, *userId*, and the full resource URI of *userId*.
- For logging out and destroying the token, use `restInstance.logout()`.

After having made sure that your application has been authenticated, it is straightforward to check if a user has a role or permission for your application or another service.

Assume that your OpenIoT application (e.g., SD&UM) is called A and has the *clientId* **serviceA** and token **tokenA**. A web application (e.g., Request-Presentation) called C with *clientId* **serviceC** requests one of your services on behalf of user U. User U has been authenticated. Token **tokenU** has been assigned to C (because in a web application, the token is given to the application

instead of the user). For fulfilling the C's request, you need to check if U is permitted to access the requested resource(s) on your application. You can write a similar code as the following:

```
String permissionString = ...
AccessControlUtil instance = AccessControlUtil.getRestInstance();
OAuthAuthorizationCredentials callerCredentials = new
OAuthAuthorizationCredentials(tokenU, serviceC, null);
OAuthAuthorizationCredentials credentials = new
OAuthAuthorizationCredentials(tokenA, serviceA, callerCredentials);
if(instance.hasPermission(permissionString, credentials)){
    // Proceed with fulfilling the request
} else {
    // Deny fulfilling the request
}
```

Now, assume that C is requesting a service from an OpenIoT service provider B (e.g., LSM), having clientId **serviceB** and token **tokenB**. For fulfilling this request, B needs to use a service from your application. For fulfilling the B's request you need to check if U is permitted to access the requested resource(s) on your application. You can write a similar code as the following:

```
String permissionString = ...
AccessControlUtil instance = AccessControlUtil.getRestInstance();
OAuthAuthorizationCredentials userCredentials = new
OAuthAuthorizationCredentials(tokenU, serviceC, null);
OAuthAuthorizationCredentials callerCredentials = new
OAuthAuthorizationCredentials(tokenB, serviceB, userCredentials);
OAuthAuthorizationCredentials credentials = new
OAuthAuthorizationCredentials(tokenA, serviceA, callerCredentials);
if(instance.hasPermission(permissionString, credentials)){
    // Proceed with fulfilling the request
} else {
    // Deny fulfilling the request
}
```

Notes:

- The same procedure can be used for testing if a user has a specific role, using `AccessControlUtil.hasRole(..)` methods.
- `OAuthAuthorizationCredentials` allows more than two callers to be chained, but this must be avoided. If the chain contains two credentials (including the credentials of the current application), the second credentials is used as user's credentials. If the chain contains three credentials (including the credentials of the current application), the third credentials is used as user's credentials. If the chain contains more than three credentials, the rest is ignored.

- In cases where you need to check if a user is authorized to access some resources on another OpenIoT service rather than your application, there are variants of the above methods that take a *targetClientId* parameter. Your application is allowed to do this only if it has the permission "**ext:retrieve_permissions**" on the target OpenIoT service provider denoted by *targetClientId*.

8.4.4.4 Sample Restful Usage

X-GSN, LSM-Light.Server, Scheduler, and SD&UM modules are configured to use the RESTful security client utilities. For each of these modules, please refer to the corresponding section in the rest of this chapter for detailed configuration and usage information.

8.4.5 Cache Configuration

In order to reduce the traffic between your application and the OpenIoT CAS, the security client can cache the authorisation information it retrieves from the OpenIoT CAS. In order to activate caching you need to add the following in the Shiro configuration file (*web-client[-MODULE_NAME].ini* or *rest-client[-MODULE_NAME].ini*).

```
cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
cacheManager.cacheManagerConfigFile=classpath:ehcache-sec-client.xml
securityManager.cacheManager = $cacheManager
```

We use Ehcache⁷ cache provider. `cacheManager.cacheManagerConfigFile` specifies the Ehcache configuration file. For example in the above configuration, Shiro will look for a file named ***ehcache-sec-client.xml*** in the classpath. A sample configuration file can be found in `security-client/src/test/resources`. Instead of the **classpath:** prefix you can use **url:**, **file:** or simply provide a valid path to the configuration file without any prefix. In Ehcache configuration file, you can change the default values in *defaultCache* element. Particularly, tune *timeToldleSeconds* and *timeToLiveSeconds* parameters.

Great attention must be given to using caching, otherwise you might give permissions to users while their tokens have been expired or they don't have those permissions anymore. You can clean the whole cache by calling:

```
AccessControlUtil.getAuthorizationManager().clearCache(null)
;
```

The authorisation information for the current user can be removed from the cache by calling `AccessControlUtil.reset()`.

8.4.6 Token Expiry

Since tokens are represented only by a string, the only way to find out if a token has been expired is to contact the OpenIoT CAS by calling `hasPermission()`

⁷ <http://ehcache.org/>

or `hasRole()` and check for an exception of type `AccessTokenExpiredException`. Since this exception is a runtime exception, you don't need to surround every call to `hasPermission()` or `hasRole()` methods by try-catch blocks. In a web application you can instruct your container to forward the user to a specific page when this exception occurs. `AccessTokenExpiredException.getToken()` returns the token that is expired.

The above approach is not reliable if we enable caching. `AccessControlUtil` also provides the method `getExpiredAccessToken(OAuthorizationCredentials credentials)`. This method returns the first expired token in credentials, if there is any. Otherwise, it returns `null`.

9 INTEGRATION OF OPENIOT COMPONENTS WITH THE SECURITY FRAMEWORK

All OpenIoT modules are integrated with the security module, in the sense that they implement access control using the means provided by the security module components. In the following we describe how this integration is done for each OpenIoT module.

9.1 LSM Server

A *rest-client.ini* file is provided in `lsm-light.server/src/main/resources/` assuming the OpenIoT CAS is deployed on *localhost*. A recommended practice is to copy this file to the JBoss configuration folder and rename it to *rest-client-lsm-server.ini*. The username and password of the user assigned to the LSM-Server module is configured in *openiot.properties* file using *security.initialize.lsmserver.username* and *security.initialize.lsmserver.password* properties.

The utility class `org.openiot.lsm.utils.SecurityUtil` is provided for authentication and authorisation tasks. It logs in the OpenIoT CAS using the username and password read from *openiot.properties* file. The token that is obtained from the security server is used when calling `AccessControlUtil.getPermission()` methods. Each request to the LSM server must contain the access token and `clientId` of the requesting application. They are retrieved from the `HttpServletRequest` object. The following code snippet shows an example of access check in the LSM Server:

```
String token = request.getHeader("token");
String clientId = request.getHeader("clientId");
String permissionString = "lsm-light.server.main:delSensor";
ServletContext context = getServletContext();
if(SecurityUtil.hasPermission(permissionString, context, token, clientId)
{
    sensorManager.sensorDelete(graphURL,infos);
} else {
    result ="You don't have permission to operate this function";
    logger.info(result);
}
```

9.2 X-GSN

A *rest-client-xgsn.ini* file is provided in `x-gsn/conf/` directory assuming the OpenIoT CAS is deployed on *localhost*. The username and password of the user assigned to the X-GSN module is configured in *application.conf* file using *username* and *password* properties.

The utility class `org.openiot.gsn.utils.CASUtils` is provided for authentication and authorisation tasks. It logs in the OpenIoT CAS using the username and password read from *application.conf*. The token that is obtained

from the security server is sent with each request to the LSM Server when publishing sensor data as the following code snippet shows.

```
LSMTripleStore lsmStore = new LSMTripleStore(lsmServer);
OAuthAuthorizationCredentials cred = CASUtils.getTokenAndId();
sensorID = lsmStore.sensorAdd(sensor, cred.getClientId(),
cred.getAccessToken());
```

9.3 Scheduler

A *rest-client.ini* file is provided in `scheduler.core/src/main/resources/` assuming the OpenIoT CAS is deployed on *localhost*. A recommended practice is to copy this file to the JBoss configuration folder and rename it to *rest-client-scheduler.ini*. The username and password of the user assigned to the Scheduler module is configured in *openiot.properties* file using *security.initialize.scheduler.username* and *security.initialize.scheduler.password* properties.

The utility class `org.openiot.scheduler.core.utils.SecurityUtil` is provided for authentication and authorisation tasks. It logs in the OpenIoT CAS using the username and password read from *openiot.properties* file. The token that is obtained from the security server is used when calling `AccessControlUtil.getPermission()` methods. This token is also sent with each request to the LSM Server as the following code snippet shows.

```
OAuthAuthorizationCredentials credentials = SecurityUtil.getCredentials();
lsmStore.pushRDF(lsmFunctionalGraph, myOntInstance.exportToTriples("N-TRIPLE"),
credentials.getClientId(), credentials.getAccessToken());
```

Other modules that call services from the Scheduler must include their access token and clientId in the request. The following code snippet shows the permission control in the `discoverSensors()` method.

```
@GET
@Path("/discoverSensors")
public SensorTypes discoverSensors(@QueryParam("userID") String userID,
@QueryParam("longitude") double longitude, @QueryParam("latitude") double latitude,
@QueryParam("radius") float radius, @QueryParam("clientId") String clientId,
@QueryParam("token") String token) {
    if(!SecurityUtil.hasPermission(PermissionsUtil.SCHEDULER_ALL, token,
clientId)){
        logger.info("Missing required permissions");
        return null;
    }
    DiscoverSensorsImpl discoverSensorsImpl = new DiscoverSensorsImpl(userID,
longitude, latitude, radius);
    return discoverSensorsImpl.getSensorTypes();
}
```


9.4 SDUM

A *rest-client.ini* file is provided in `sdum.core/src/main/resources/` assuming the OpenIoT CAS is deployed on *localhost*. A recommended practice is to copy this file to the JBoss configuration folder and rename it to *rest-client-sdum.ini*. The username and password of the user assigned to the SD&UM module is configured in *openiot.properties* file using *security.initialize.sdum.username* and *security.initialize.sdum.password* properties.

The utility class `org.openiot.sdum.core.utils.SecurityUtil` is provided for authentication and authorisation tasks. It logs in the OpenIoT CAS using the username and password read from *openiot.properties* file. The token that is obtained from the security server is used when calling `AccessControlUtil.getPermission()` methods. Other modules that call services from the SD&UM must include their access token and `clientId` in the request. The following code snippet shows the permission control in the `pollForReport()` method.

```
@GET @Path("/pollforreport")
public SdumServiceResultSet pollForReport(@QueryParam("serviceID") String
applicationID, @QueryParam("clientId") String clientId, @QueryParam("token") String
token) {
    if(!SecurityUtil.hasPermission(PermissionsUtil.SDUM_ALL, token, clientId)){
        logger.info("Missing required permissions");
        return null;
    }
    PollForReportImpl pollForReportImpl = new PollForReportImpl(applicationID);
    return pollForReportImpl.getSdumServiceResultSet();
}
```

9.5 Request Definition

A *web-client.ini* file is provided in `ui.requestDefinition/src/main/resources/` assuming the OpenIoT CAS is deployed on *localhost*. A recommended practice is to copy this file to the JBoss configuration folder and rename it to *web-client-requestDefinition.ini*. While accessing the main page of the request definition web application, if the user is not logged in, she will be redirected to the login page provided by the OpenIoT CAS. The code snippet below shows how this is done.

```
AccessControlUtil acUtil = AccessControlUtil.getInstance();
if(acUtil.getOAuthAuthorizationCredentials() == null){
    HttpServletResponse response = (HttpServletResponse)
FacesContext.getCurrentInstance().getExternalContext().getResponse();

    HttpServletRequest req = (HttpServletRequest)
FacesContext.getCurrentInstance().getExternalContext().getRequest();

    acUtil.redirectToLogin(req, response);
}
```

When calling a service from the Scheduler, the web application adds the user token and its clientId to the request parameters as the code below shows.

```
AccessControlUtil acUtil = AccessControlUtil.getInstance();
OAuthAuthorizationCredentials creds = acUtil.getOAuthAuthorizationCredentials();
ClientRequestFactory factory = new ClientRequestFactory(SCHEDULER_HOST_URL);
ClientRequest request =
factory.createRelativeRequest("/rest/services/getAvailableApps");
request.queryParameter("userID", creds.getUserIdURI());
request.queryParameter("clientId", creds.getClientId());
request.queryParameter("token", creds.getAccessToken());
...
```

9.6 Request Presentation

A *web-client.ini* file is provided in `ui.requestPresentation/src/main/resources/` assuming the OpenIoT CAS is deployed on *localhost*. A recommended practice is to copy this file to the JBoss configuration folder and rename it to *web-client-requestPresentation.ini*. While accessing the main page of the request presentation web application, if the user is not logged in, she will be redirected to the login page provided by the OpenIoT CAS. The code snippet below shows how this is done.

```
AccessControlUtil acUtil = AccessControlUtil.getInstance();
if(acUtil.getOAuthAuthorizationCredentials() == null){
    HttpServletResponse response = (HttpServletResponse)
FacesContext.getCurrentInstance().getExternalContext().getResponse();

    HttpServletRequest req = (HttpServletRequest)
FacesContext.getCurrentInstance().getExternalContext().getRequest();

    acUtil.redirectToLogin(req, response);
}
```

When calling a service from the SD&UM, the web application adds the user token and its clientId to the request parameters as the code below shows.

D5.2.2 Privacy and Security Framework b

```
AccessControlUtil acUtil = AccessControlUtil.getInstance();
OAuthAuthorizationCredentials creds = acUtil.getOAuthAuthorizationCredentials();
ClientRequestFactory factory = new ClientRequestFactory(SDUM_HOST_URL);
ClientRequest request =
factory.createRelativeRequest("/rest/services/getAvailableApps");
request.queryParameter("serviceID", serviceID);
request.queryParameter("clientId", creds.getClientId());
request.queryParameter("token", creds.getAccessToken());
...
```

10 CONCLUSIONS

In this document we presented the foundations of the security/privacy framework in OpenIoT. The main feature of the security/privacy architecture is the use of the Central Authorisation Server (CAS) for authorizing applications running on behalf a user by granting them an access token with a given time to live. This prevents any circulation of the credentials throughout OpenIoT components.

We have also provided an overview of the currently implemented prototype. The documentation for how to use and configure the prototype can be found on the OpenIoT wiki. We have described how we adapted Jasig CAS for authentication and authorisation, which allows flexible multi-protocol Single Sign-On (SSO) capabilities. We have shown how CAS can be configured to act as an OAuth2.0 server, which we referred to as *OpenIoT CAS*. We also explained the internals of the other two modules of the security framework, namely the *Management Console*, and *Security Client*. The overall authentication and authorisation approach in each module is explained in full details.

Finally, we have detailed how all OpenIoT modules have been integrated with the security module, in the sense that they implement access control using the means provided by the security module components. A complete description has been provided for each core module.

This document presents the second and final part of the security/privacy specification (deliverable).

11 REFERENCES

- [1] RFC 4303 IP Encapsulating Security Payload ESP
- [2] [RFC 5246](#), The Transport Layer Security (TLS) Protocol.
- [3] [RFC 2818](#), HTTP Over TLS.
- [4] [RFC 5849](#), The OAuth 1.0 Protocol.
- [5] [RFC 6749](#), The OAuth 2.0 Authorisation Framework.
- [6] Ravi S. Sandhu, "Lattice-Based Access Control Models". *The Computer Journal*, Volume 26, Issue 11, November 1993, Page 9-19.
- [7] Hyo-Sankg Lim, Yang-Sae Moon and Elisa Bertino, "Provenance-based Trustworthiness Assessment in Sensor Networks", *DMSN'10*, September 13, 2010, Singapore.
- [8] H.-S. Lim, G. Ghinita, E. Bertino, and M. Kantarcioglu. A game-theoretic approach for high-assurance of data trustworthiness in sensor networks. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1192–1203, Washington, DC, USA, 1-5 April 2012.
- [9] Y. Ma, Y. Guo, X. Tian, and M. Ghanem. Distributed clustering-based aggregation algorithm for spatial correlated sensor networks. *IEEE Sensors Journal*, 11(3):641–648, 2011.
- [10] M. C. Vuran, O. B. Akan, and I. F. Akyildiz. Spatio-temporal correlation: theory and applications for wireless sensor networks. *Computer Networks Journal* (Elsevier, 45:245–259, 2004.
- [11] R. Gwadera. Multi-stream join answering from mining significant cross-stream correlations. *Frontiers of Computer Science*, 6(2):131–142, 2012.
- [12] Eran Hammer. OAuth. <http://hueniverse.com/oauth/>
- [13] Brad Rubin, Java security: Java security, Part 1: Crypto basics. <http://www.ibm.com/developerworks/java/tutorials/j-sec1/section7.html>
- [14] Java™ Cryptography Architecture (JCA) Reference Guide <http://download.java.net/jdk7/archive/b125/docs/technotes/guides/security/crypto/CryptoSpec.html>
- [15] Vaudenay, Serge. A classical introduction to cryptography: Applications for communications security. Springer, 2006.
- [16] Rivest, R.; A. Shamir; L. Adleman (1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM* **21** (2): 120–126.
- [17] Menezes, A. J.; van Oorschot, P. C.; Vanstone, Scott A. (1997). Handbook of Applied Cryptography.
- [18] Delfs, Hans & Knebl, Helmut (2007). "Symmetric-key encryption". *Introduction to cryptography: principles and applications*. Springer.

- [19] Biham, Eli and Shamir, Adi (1991). "Differential Cryptanalysis of DES-like Cryptosystems". *Journal of Cryptology* **4** (1): 3–72.
- [20] Joan Daemen, Vincent Rijmen, "The Design of Rijndael: AES – The Advanced Encryption Standard." Springer, 2002.
- [21] Diffie, W.; Hellman, M. (1976). "New directions in cryptography". *IEEE Transactions on Information Theory* **22** (6): 644–654.
- [22] keytool - Key and Certificate Management Tool
<http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html>
- [23] T. Dierks, E. Rescorla (August 2008). "The Transport Layer Security (TLS) Protocol, Version 1.2"
- [24] A. Freier, P. Karlton, P. Kocher (August 2011). "The Secure Sockets Layer (SSL) Protocol Version 3.0".
- [25] Dieter Gollmann. Computer Security, 3rd Edition. December 2010. Wiley.
- [26] Rescorla, Eric. *SSL and TLS: designing and building secure systems*. Vol. 1. Reading: Addison-Wesley, 2001.
- [27] Sandhu, Ravi S., et al. "Role-based access control models." *Computer* 29.2 (1996): 38-47.
- [28] Sandhu, Ravi S., and Pierangela Samarati. "Access control: principle and practice." *Communications Magazine, IEEE* 32.9 (1994): 40-48.
- [29] Ferraiolo, David, Janet Cugini, and D. Richard Kuhn. "Role-based access control (RBAC): Features and motivations." *Proceedings of 11th annual computer security application conference*. 1995.
- [30] Moffett, Jonathan, Morris Sloman, and Kevin Twidle. "Specifying discretionary access control policy for distributed systems." *Computer Communications* 13.9 (1990): 571-580.
- [31] Osborn, Sylvia. "Mandatory access control and role-based access control revisited." *Proceedings of the second ACM workshop on Role-based access control*. ACM, 1997.